UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Valentin Kragelj

Pregled in analiza tehnologij za serializacijo objektov

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2014

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Valentin Kragelj

Pregled in analiza tehnologij za serializacijo objektov

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: viš. pred. dr. Marko Privošnik

Ljubljana, 2014

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Serializacija igra pomembno vlogo pri razvoju in delovanju sodobnih računalniških sistemov. Poleg njene vloge se je v preteklosti povečevalo tudi število različnih tehnologij za serializacijo. Tako danes obstajajo številne bolj ali manj različne tehnologije in formati, ki omogočajo serializacijo podatkov. V okviru diplomske naloge predstavite vlogo serializacije v računalniških sistemih ter analizirajte in primerjajte izbrano množico značilnih tehnologij za serializacijo. V zaključku diplomske naloge podajte ugotovitve in priporočila za uporabo tehnologij za serializacijo.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Valentin Kragelj, z vpisno številko **63100205**, sem avtor diplomskega dela z naslovom:

Pregled in analiza tehnologij za serializacijo objektov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom viš. pred. dr. Markota Privošnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Vrtojbi, dne 14. septembra 2014 Podpis avtorja:

Zahvaljujem se moji družini za podporo v času študija ter mentorju viš. pred. dr. Markotu Privošniku za vso pomoč pri izdelavi diplomske naloge.

Kazalo

Kazalo slik

Kazalo tabel

Kazalo grafikonov

Povzetek

V diplomskem delu je podan pregled tehnologij za serializacijo objektov. Serializacija je prestrukturiranje objektov v obliko, ustrezno za prenos ali shrambo. S serializacijo poskušamo rešiti probleme, ki nastanejo pri prenosu obsežnih podatkov med sistemi. Ti problemi so lahko prostorska (in posledično časovna) zahtevnost prenosa podatkov in odvisnost zapisa podatkov od okolja v katerem so bili ustvarjeni.

Na voljo je veliko rešitev, za katere je značilno, da želijo poenostaviti in pohitriti delo s podatki s pomočjo procesa serializacije. Pri tem so nekatere rešitve boljše za neke probleme, nekatere manj. V diplomski nalogi sem pregledal nekaj tehnologij za serializacijo objektov, jih opisal in testiral ter na podlagi njihovih lastnosti in praktične uporabe predlagal najboljše oziroma najustreznejše za različne situacije. Rezultati so bili pričakovani; rešitve ki temeljijo na binarnih formatih so časovno hitrejše ter prostorsko manj požrešne, a so podatki človeku manj razumljivi.

Ključne besede: serializacija, deserializacija, testiranje, formati.

Abstract

The thesis provides an overview of technologies for serializing objects. Serialization is process of restructuring object for transfering or storing data. The serialization tries to solve problems that arise when dealing with large-scale transfers of data between systems. These problems can be spatial (and hence time) complexity for transfering data and dependence of data from the environment in which they were generated.

There are a lot of solutions. Their goals are to simplify and speed up the work with data through a process of serialization. In doing so, some solutions are better for some problems, amd some less. In this thesis, I examined some of the technologies for object serialization, described and tested them, and on the basis of their properties and practical application, proposed best or most appropriate for different situations. The results were expected; solutions based on binary formats are faster and spatially less greedy, but also harder to be understood by human.

Keywords: serialization, deserialization, testing, formats.

Poglavje 1

Uvod

Večina današnjih programov ima tako ali drugače opravka s prenosljivostjo podatkov med programskimi sistemi in/ali shranitve podatkov v podatkovno zbirko. Ker je teh podatkov lahko ogromno, se je že v zgodnjih (računalniških) letih pojavil problem preobsežnosti podatkov. V slednjem primeru se lahko zgodi, da ima podjetje več stroškov za nakup opreme (računalnikov, trdih diskov, računalniškega pomnilnika, itd.) kot bi bilo potrebno. Ker serializirani podatki zasedejo manj prostora, posledično potrebuje podjetje manj opreme za shrambo teh podatkov. Serializacija je proces manipulacije objekta v obliko, ki je primerna za prenos preko omrežja ali za shranitev na prenosni medij, pri čemer so podatki lahko zapisani bolj strnjeno (sicer odvisno od posamezne tehnološke rešitve) in v obliki, ki jo razumejo tudi drugi sistemi.

V poznih osemdesetih letih je podjetje Sun Microsystems (katerega danes poznamo predvsem po programskem jeziku Java) ustvarilo prvi format za serializacijo podatkov – XDR ali External Data Representation [2]. Kmalu se je ustvarila potreba po alternativnih formatih, saj XDR ni zadostoval za vse vrste podatkov. V poznih devetdesetih letih je tako nastal XML, danes najbolj razširjen in poznan format, ki serializira podatke v tekstovni, človeku razumljivi obliki. Kmalu je sledilo še veliko ostalih formatov, kot so JSON, YAML, Protocol Buffers in ostali, veliko programskih jezikov (Java, PHP, Python, Ruby, …) pa je celo implementiralo lastne rešitve za serializacijo objektov. Namen procesa serializacije je ohranitev stanja objekta, kloniranje objekta ali prenos objekta v drug sistem.

Danes je na voljo veliko preveč formatov, da bi se lahko zlahka odločili kateri je najbolj primeren za določeno situacijo. Zgolj bolj popularnih je okoli trideset, pri čemer je vsak dober za neko situacijo in slabši v drugačnih pogojih.

Za izdelavo diplomske naloge na področju serializacije objektov sem se odločil na podlagi dela, ki sem ga opravljal na delovni praksi v zadnjem letniku dodiplomskega študija. Tam sem delal z raznimi formati za serializacijo, pri čemer sem imel težave z izbiro ustreznih formatov. S to diplomsko nalogo želim jasno predstaviti nekaj formatov ter njihovo praktično uporabnost v industriji.

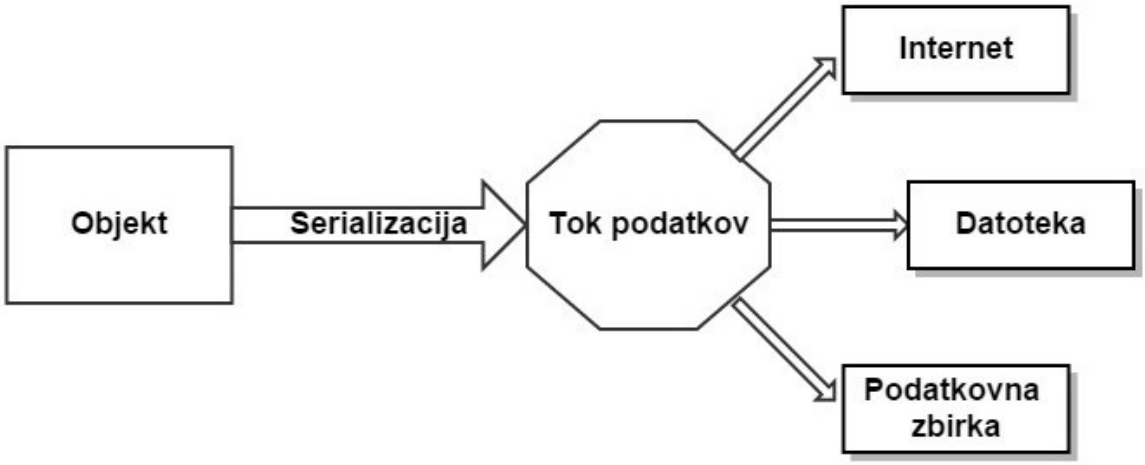
V diplomski nalogi sem se moral omejiti na najpopularnejše formate, pri čemer sem poskušal izbrati tudi najbolj raznovrstne. To pomeni, da nisem izbral formatov, ki so si med seboj zelo podobni po namembnosti oziroma ciljih, a so vseeno dovolj razširjeni, da jih je praktično implementirati. Seznamu formatov sem dodal še podporo serializaciji v Javi, da sem povzel tudi interno podporo serializaciji v programskih jezikih.

V prvem delu diplomske naloge sem opisal pomen serializacije in deserializacije ter lastnosti le-teh. Sledijo opisi formatov ter za večino njih tudi primer zapisa podatkov v pripadajočem formatu. Primeri so med seboj pomensko enakovredni z namenom lažje primerjave oblike podatkov v teh formatih. V zadnjem delu sem formate primerjal med seboj ter opisal, kje se v praksi uporabljajo. Za določene formate sem v zadnjem delu opravil tudi teste v povezavi s hitrostjo serializiranja in deserializiranja podatkov in kompaktnostjo zapisanih podatkov. V zaključku sem napisal sklepne ugotovitve ter priporočila, v kakšnih pogojih uporabiti kateri format.

Poglavje 2

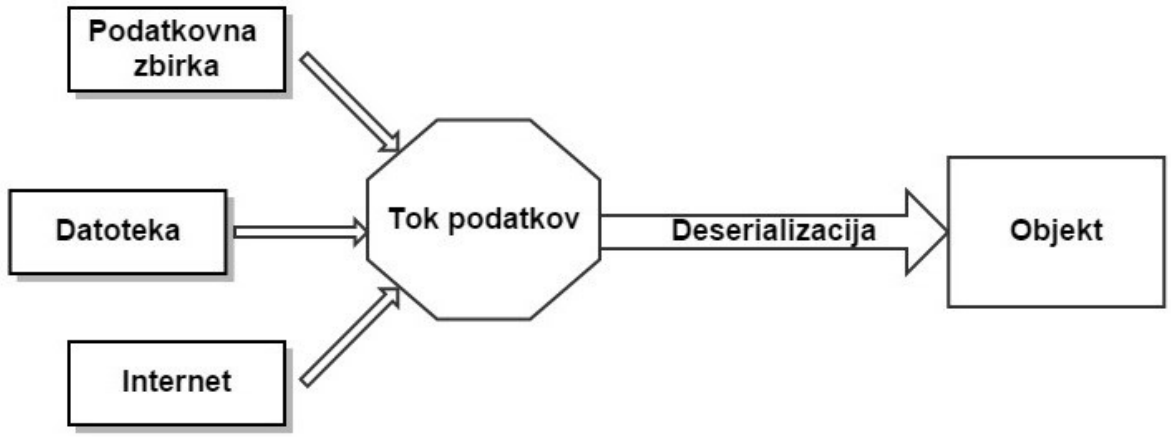
Serializacija ter deserializacija

Serializacija je proces obdelave objekta v tok podatkov (bajtov) v obliko, ki je primerna za shranjevanje ali prenos [1]. Shrani se ga lahko v podatkovno zbirko, računalniški pomnilnik, datoteko ali pa ga prenesemo preko omrežja. Cilj tega početja je, da shranimo stanje objekta z namenom kasnejše uporabe ali uporabe v drugem sistemu. Slika 2.1 prikazuje proces serializacije.



Slika 2.1: Diagram poteka serializacije.

Deserializacija je obraten postopek serializacije. Tu tok podatkov (v kateri je shranjen objekt, ki je bil prej serializiran) preberemo ter ustvarimo shranjen objekt v delovnem sistemu. Slika 2.2 prikazuje proces deserializacije.



Slika 2.2: Diagram poteka deserializacije.

Prednosti serializacije:

- shranjevanje podatkov in stanja objekta,
- prenos podatkov oziroma stanja objekta (na primer preko omrežja),
- kloniranje objekta,
- možnost porazdelitve objektov na različnih sistemih,
- preverjanje različnih stanj objekta.

Za nekatere od zgornjih lastnosti (na primer porazdelitev objektov) je potrebna neodvisnost arhitekture. To pomeni, da mora računalnik z drugačno programsko opremo pravilno rekonstruirati tok podatkov v objekt.

Pomembna lastnost serializiranih objektov je nezmožnost deserializiranja le dela toka podatkov. To pomeni, da če hočemo prebrati le določene podatke iz serializiranega objekta, moramo imeti celoten tok podatkov (tj. serializiran objekt), ga deserializirati v celoten objekt ter šele nato lahko preberemo določen atribut.

Slabosti serializacije:

- postopek serializacije in deserializacije je lahko časovno zahteven za procesor ter prostorsko zahteven za računalniški pomnilnik ali prenosni medij, še posebej če so objekti veliki ali jih je številčno veliko,
- pri spreminjanju objektov lahko pride do težav pri kompatibilnosti,
- osnovna serializacija serializira vse atribute objekta, tudi privatne (na primer gesla), razen če specifično določimo katere ne želimo serializirati.

Serializacija je torej uporabna, ko želimo podatke shraniti ali jih prenesti iz mej delovnega okolja. Navadno se proces uporablja pri prenosu podatkov med sistemom in spletnimi storitvami. Pozorni pa moramo biti na slabosti serializacije.

2.1 Formati za serializacijo

Leta 1987 je podjetje Sun Microsystems objavilo XDR format. XDR je standardni format za serializacijo podatkov, ki omogoča prenos podatkov med dvema različnima računalniška sistema. Ko XDR ni bilo več optimalno uporabljati za vse vrste podatkov, se je kot alternativa temu standardu leta 1998 pojavil format XML, kmalu zatem pa še JSON. Ta dva formata sta v spadat v današnjem svetu med najbolj priljubljene. Nekateri drugi formati so še YAML, Apache Avro, NextSTEP, GNUstep, ASN, BSON, MessagePack ter Thrift. O nekaterih bomo spregovorili v naslednjem poglavju.

2.1.1 Vrste formatov za serializacijo

Formate ločimo glede na dva kriterija - ali je za podatke potrebno definirati shemo ter ali so serializirani podatki binarizirani ali ne.

Če format uporablja shemo, to pomeni, da mora programer definirati kakšne tipe podatkov bo serializiral. Cilj uporabe sheme je hitrejši proces serializacije in deserializacije ter kompaknost zapisa podatkov. Slaba stran uporabe sheme je, da je s strani programerja potrebno več dela za implementacijo formata ter dejstvo, da je omejen na vrsto podatkov, katere je v začetku definiral oziroma opisal v shemi. Primer formata, ki potrebuje shemo, je v nadaljevanju opisani Protocol Buffers. Pri njem se na začetku ustvari tako imenovano proto datoteko, v kateri definiramo strukturo podatkov.

Formate lahko razdelimo tudi na to, ali so binarni ter nebinarni oziroma tekstovni. Za binarne formate je značilno, da so podatki kodirani v binarni obliki (tj. obliki, ki je razumljiva računalniku), medtem ko podatki v nebinarnih formatih predstavljeni v tekstovni, človeku razumljivi obliki.

Primer podatkov, predstavljenih v tekstovni obliki (format JSON):

```
{"hello": "world"}
```

Primer enakih podatkov, predstavljenih v formatu BSON, ki je binaren format (podatki so zapisani skladno s Pythonovo sintakso):

```
\x16\x00\x00\x00\x02hello\x00\x06\x00\x00\x00world\x00\x00
```

Iz binarnega zapisa ne moremo prebrati, kakšni podatki so serializirani, medtem ko so podatki v tekstovni obliki razumljivi.

Cilj binariziranja podatkov je hitrejši proces serializacije in deserializacije ter kompaktnost zapisa podatkov, saj so podatki že v procesorju razumljivi obliki.

2.1.2 Podpora serializaciji v programskih jezikih

Veliko objektno usmerjenih programskih jezikov ima implementirano lastno podporo za serializacijo objektov z uvedbo vmesnika (tj. predpripravljena orodja, posebej namenjena serializaciji). Ti jeziki so na primer Ruby, Python, PHP, Objective-C, Java, .NET, Perl ter R.

Cilji interne podpore serializacije v programskih jezikih so lažje in hitrejše delo, saj programerju ni potrebno dodajati dodatnih vmesnikov ali knjižnic za serializiranje.

Poglavje 3

Formati za serializacijo

3.1 XML

XML ali Extensible Markup Language (razširljiv označevalni jezik) je preprost ter fleksibilen označevalni jezik, ki izvira iz SGML (Standard Generalized Markup Language – standard za markiranje dokumentov) [6]. Namen XMLja je kodiranje dokumentov v formatu, ki je razumljiv ljudem in računalniku. Sprva je bil zasnovan na področju elektronskega založništva, sčasoma se je začel uveljavljati tudi pri izmenjavi podatkov po medmrežju, lahko pa se ga uporablja tudi za predstavitev podatkovnih struktur (čeprav za to obstajajo boljši formati). Datoteke XML uporabljajo končnico ».xml«.

Cilj XMLja je enostavnost, splošnost ter uporabnost na spletu. Uporablja format Unicode, zaradi česar je primeren za različne (človeške) jezike.

XML je spada med najbolj popularne formate. Jeziki kot so RSS, SOAP, XHTML in drugi, kot tudi veliko programov – Microsoft Office, OpenOffice, LibreOffice ter iWork – uporabljajo XML sintakso. Prav tako je popularen v komunikacijskih protokolih, kot na primer XMPP (Extensible Messaging and Presence Protocol). XML je trenutno v splošni rabi za izmenjavo podatkov preko spleta in eden najpopularnejših formatov.

Format Unicode je standard za kodiranje znakov v računalniku [5]. Je sistem, ki na osnovi tabele predstavi črke kot kode, ki so razumljive za računalnik. Primer (dela) tabele:

Koda znaka	**Človeku razumljiv zapis**	**Decimalen zapis**	**Opis**
U+0061	a	&#97	Latinska mala črka A
U+0062	b	&#98	Latinska mala črka B
U+0063	c	&#99	Latinska mala črka C

Tabela 3.1: Primeri znakov v formatu Unicode.

Oznaka in vsebina

XML dokument se v osnovi deli na oznake ter vsebino. Oznake prepoznamo po posebnih znakih »<« za začetek oznake ter »>« za konec oznake. Besedilo, ki ne spada pod oznako, definiramo kot vsebina.

Oznaka

Poznamo 3 vrste oznak:

- začetna oznaka (primer: `<ime>`, postavi se jo pred ustrezno vsebino),
- končna oznaka (primer: `</ime>`, postavijo se jo za ustrezno vsebino,
- prazna oznaka (primer: `<ime/>`, pomeni da oznaka nima vsebine).

Element

Je del XML dokumenta, ki je sestavljen iz začetne oznake, vsebine ter končne oznake ali je zgolj prazna oznaka. Namesto vsebine lahko tudi gnezdimo več elementov. Primer elementa:

```
<ime>Valentin</ime>.
```

Atribut

Je lahko del začetne ali prazne oznake in se uporablja večinoma za metapodatke elementa, v katerem se nahaja. Atributov ni mogoče gnezditi. Primer atributa:

```
<višina enota="meter">187</višina>
```

XML deklaracija

Namen deklaracije je, da se dokument prepozna kot XML dokument. Deklaracijo se definira tako, da se v prvo vrstico XML datoteke vpiše naslednje podatke:

```
<?xml version= "1.0" encoding= "UTF-8">
```

Zgornji kodi lahko rečemo tudi prolog.

Komentar

Če želimo napisati komentar v kodi, uporabimo naslednjo strukturo:

```
<!-- tu vpišemo komentar, ki je lahko poljubno besedilo -->
```

Kot vidimo, potrebujemo le »<!—» na začetku ter »-->« na koncu.

Primer dokumenta v formatu XML je sledeč:

Koda 1: Primer XML dokumenta.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <oseba>
3.       <ime>Valentin</ime>
4.       <starost>23</starost>
5.       <seznam>
6.            <element id="1">FRI</element>
7.            <element id="2">VSŠ</element>
8.       </seznam>
9.       <slovar>
10.           <element kljuc="ulica">Pod Lazami</element>
11.           <element kljuc="stevilka">32</element>
12.      </slovar>
13. </oseba>
```

Ustvarili smo XML dokument z opisom osebe. Določili smo ji elemente ime, starost, seznam (ki vsebuje dva elementa) in slovar (ki vsebuje dva elementa). Opazimo, da se števila in nize v vsebini elementov zapiše v dvojnih narekovajih.

Posebni znaki

Ker XML simbole kot so na primer »<«, »>«, »&« obravnava kot del oznake in ne vsebino, jih moramo zapisati drugače, če želimo, da so del vsebine. Vzemimo za primer besedilo »Jaz <3 tebe« (kar se lahko zapiše »Jaz ljubim tebe« oz. »Jaz te ljubim«). Če ga zapišemo tako, bo sistem, ki bere XML dokument, prebral znak »manjše kot« (<) kot začetek oznake ter bo javil napako, saj dokument ne bo sintaktično pravilen. Namesto omenjenega znaka moramo uporabiti poseben znak »&lt« (lt je kratica za less than – manjše kot).

Primer se torej mora zapisat kot »Jaz &lt3 tebe«. Primer uporabe primera bi tako bil lahko:

Koda 2: Primer XML dokumenta s posebnim znakom.

```
1.  <sporočilo>
2.       <pošiljatelj>Romeo</pošiljatelj>
3.       <prejemnik>Julia</prejemnik>
4.       <besedilo>Jaz &lt3 tebe</besedilo>
5.  </sporočilo>
```

Pravila sintaktično pravilnega XML dokumenta

XML dokument je ustrezno formiran, če:

- so elementi pravilno ugnezdeni,
- imajo atributi unikatna imena,
- vsebuje le znake, definirane v Unicode formatu,
- ima začetna oznaka tudi pripadajočo končno oznako,
- so imena oznak smiselna glede na vsebino, ki jo vsebujejo,
- vsebuje le en korenski element.

Razčlenjevalniki

XML dokument lahko manipuliramo tudi programsko. Dva osnovna tipa razčlenjevalnikov (ang. parser) so [7]:

- SAX (Simple API for XML)
- DOM (Document Object Model)

SAX je Javin programski vmesnik (Java API oziroma application program interface). Deluje kot dogodkovno voden vmesnik, ki prebere dokument serijsko, vsebina dokumenta pa je podana ob klicih metod. SAX torej bazira na dogodkih.

DOM (document object model) je prav tako programski vmesnik, le da bazira na drevesu objektov XML dokumenta. To pomeni, da loči vse elemente v XML dokumentu in za vsakega ustvari vozlišče. Do vsakega elementa lahko nato dostopamo neovirano, možno je pa tudi spreminjanje vsebine elementov. XML dokument lahko ustvarimo z uporabo DOM vmesnika v jeziku Java na slednji način:

Koda 3: kreiranje XML dokumenta z uporabo DOMa.

```
1.  DocumentBuilderFactor fact =
    DocumentBuilderFactory.newInstance()
2.  DocumentBuilder parser = fact.newDocumentBuilder()
3.  Document xml = parser.newDocument()
4.  Node oseba = xml.createElement("oseba")
5.  xml.appendChild(oseba)
6.  Node ime = xml.createElement("ime")
7.  oseba.appendChild(ime)
8.  ime.appendChild(xml.createTextNode("Valentin"))
```

V prvih treh vrsticah kode 3 pripravimo nov XML dokument z imenom »xml«. Nato v četrti vrstici ustvarimo nov element z oznako »oseba«, katero nato v peti vrstici vstavimo v XML

dokument. V šesti vrstici prav tako ustvarimo nov element, tokrat z oznako »ime«. Tega nato pripnemo (ugnezdimo) v element oseba. Na koncu pripnemo elementu ime vsebino, ki je besedilo »Valentin«. Izgled XML dokumenta je torej sledeč:

Koda 4: Ustvarjen XML dokument.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <oseba>
3.       <ime>Valentin</ime>
4.  <oseba>
```

Opazimo še, da se deklaracija dokumenta ustvari samodejno v DOM razčlenjevalniku.

3.2 JSON

JSON (JavaScript Objet Notation) je odprt format za izmenjavo strukturiranih podatkov preko omrežja [9]. Je standardiziran format, ki se ga večinoma uporablja kot alternativo XML formatu. Tako kot XML je tudi JSON človeku razumljiv format ter neodvisen od platforme. Podatki za izmenjavo so najpogosteje iz parov atributov in vrednosti. Format se veliko uporablja za uvoz oziroma izvoz podatkov, shranjevanje konfiguracije orodij in shranjevanje nastavitev skript.

Za končnico JSON datotek se uporablja ».json«.

Ker je format standardiziran, ga podpira oziroma uporablja veliko programskih jezikov ter programov. Med drugimi so to C, C++, Java, Delphi, Objective C, C#, PHP, Photoshop in drugi.

Lastnosti formata JSON:

- lahek za uporabo,
- hiter ter kompakten,
- uporaben za urejene sezname ter za pare atributov in vrednosti,
- neboleča uporaba v programskih jezikih.

Tipi objektov v formatu JSON

V JSONu lahko definiramo naslednje tipe:

- število (ni pomembno, ali je zapisano kot celo število ali decimalka),
- niz (zaporedje črk po tabeli Unicode, lahko tudi prazen niz),
- logična vrednost (bodisi true, bodisi false),
- tabela (zaporedje nič ali več elementov, za katere ni potrebno, da so enakega tipa. Tabelo se označi z oglatimi oklepaji),
- slovar (podobno tabeli, le da je namesto elementa par ključa in vrednosti. Označi se ga z zavitimi oklepaji),
- null (prazna vrednost.

Zapis pomensko enakovrednega primera v formatu JSON je sledeč:

Koda 5: Primer JSON dokumenta.

```
1.  {
2.      "ime": "Valentin",
3.      "starost": 23,
4.      "seznam": [
5.          "FRI",
6.          "VSŠ"
7.      ],
8.      "slovar": {
9.          "ulica": "Pod Lazami",
10.         "stevilka": 32
11.     }
12. }
```

Kot lahko opazimo je koda človeku razumljiva. Besedilo in polja označimo z dvojnimi narekovaji, medtem ko števila format sam prepozna.

3.3 BSON

BSON (Binary JSON) je format za izmenjavo podatkov. Uporablja se ga za binarno serializacijo dokumentov tipa JSON. BSON nadgradi JSON z razširitvami, ki omogočajo predstavitev dodatnih podatkovnih tipov, kot so Date type (objekt tipa datum) [10].

BSON je bil zasnovan z naslednjimi lastnostmi v mislih [4]:

- minimalističen,
- prenosljiv,
- učinkovit.

V naslednjem primeru je prikazano funkcija, kako BSON nadgradi JSON v smislu učinkovitosti.

```
["nepomembno besedilo",
"še eno besedilo",
"geslo je puran"]
```

Ko serializiramo zgornji primer v format BSON, bo le-ta dodal pred vsak element velikost le-tega v obliki številke.

```
[19 "nepomembno besedilo",
15 "še eno besedilo",
14 "geslo je puran"]
```

S to razširitvijo dosežemo večjo učinkovitost pri razčlenjevanju (parsanju) podatkov, saj razčlenjevalnik točno ve koliko je velik posamezen element, kar pomeni, da lahko po možnosti preskoči element. Če tega podatka ne bi imel, bi moral preverjati vsak znak, da izve ali je prišel do konca trenutnega elementa [11].

BSONa na koncu podakte še binarizira. Zgolj za predstavo prikažemo to v naslednjem primeru. Naslednji slovar:

```
{"hello": "world"}
```

se spremeni v (zapisan skladno z Pythonovo sintakso):

```
\x16\x00\x00\x00\x02hello\x00 \x06\x00\x00\x00world\x00\x00
```

3.4 YAML

YAML (rekurzivna kratica za YAML Ain't Markup Language) je še en izmed formatov za serializacijo podatkov v človeku prijazni, tekstovni obliki ter namenjen vsem programskim jezikom [12]. Koncept si je sposodil iz programskih jezikov C, Perl ter Python.

Sintaksa formata YAML je zasnovana z namenom hitre preslikave podatkov v podatkovne tipe, kot so na primer seznam in tabela. Človeku prijazna sintaksa je uporabna, če želi človek sam pregledati in spreminjati podatke v datoteki. Pri tem je pomembna lastnost formata, da zapis ne vsebuje narekovajev, oglatih oklepajev ter ostalih znakov, ki otežujejo branje.

Po zgradbi je YAML zelo podoben formatu JSON. Razlika je v ciljih formata: medtem ko se JSON usmerja v preprostost in univerzalnost, je cilj YAMLa še bolj razumljiva sintaksa ter boljša podpora podatkovnim strukturam.

Sintaksa formata YAML

Seznam definiramo na dva načina []. S pomišljajem ter presledkom, nato sledi besedilo:

```
- XML
- JSON
- YAML
```

ali z uporabo oglatih oklepajev:

```
[XML, JSON, YAML]
```

Slovar

Če želimo zapisati podatke kot par ključev in atributov, uporabimo naslednji način.

```
name: Valentin Kragelj
age: 23
```

Oziroma z drugačnim zapisom:

```
{name: Valentin Kragelj, age: 23}
```

Besedilo

Besedila se lahko zapiše na dva načina. Prvi ohrani prehode v novo vrstico, medtem ko drugi spremeni prehod v novo vrstico v presledek.

```
|
To besedilo bo ohranilo prehode v novo vrstico


>
To besedilo bo namesto prehodov
v novo vrstico
uporabilo presledek.
```

Komentar

Če želimo dodati komentar, uporabimo znak »#«.

```
# tu lahko vpišemo poljuben tekst
```

Zapis primera v YAML formatu je sledeč (Koda 6).

Koda 6: Primer YAML dokumenta.

```
1.  ---
2.  ime: Valentin
3.  starost: 23
4.  seznam: [FRI, VSŠ]
5.  slovar: {ulica: Pod Lazami, stevilka: 32}
6.  Ponavljajoči deli
```

Če želimo določen del YAML dokumenta uporabiti večkrat, uporabimo znak »&«, za ponovno uporabo označenega dela pa zvezdico (Koda 7).

Koda 7: Ponavljanje delov v YAML dokumentu.

```
1.  ---
2.  - korak: &id01
3.    naloga: spanje
4.
5.  - korak: &id02
6.    naloga: igranje
7.
8.  - korak: *id01
9.  - korak: *id02
10. - korak: *id01
11. - korak: *id02
```

V osmi vrstici smo namesto ponovnega pisanja celotnega koraka (kot smo ga naredili v tretji vrstici), uporabili kar referenco na že opisan korak.

Pomembna je še prva vrstica, torej trije pomišljaji. Ti definirajo začetek YAML dokumenta ter so nujno potrebni, da se dokument lahko bere kot YAML.

Podatkovni tipi

YAML je zmožen sam prepoznati tip vnesenih podatkov, če želimo specifično določiti tip, pa naredimo to na naslednji način:

ekspliciten niz: `!!str 123`

eksplicitno decimalno število: `!!float 123`

3.5 MessagePack

MessagePack je format za serializacijo podatkov, ki je uporaben predvsem pri enostavnih podatkovnih strukturah, kot so tabele, števila ter besedila. Cilj formata je bazirati na formatu JSON [13], a ga izboljšati v kompaktnosti podatkov in hitrosti serializacije. MessagePack podatke binarizira, da je lahko čim bolj kompakten, hiter ter majhen. Podpira ga veliko jezikov, med drugimi C, C++, C#, Java, javaScript, Perl, PHP, Python ter drugi.

MessagePack lahko shrani različne podatkovne tipe:

- nil (ali Nan – not a number),
- logično vrednost (true ali false),
- celo število,
- decimalno število,
- tabelo,
- slovar.

Ker se format specializira na čim manjšo velikost ter kompaktnost podatkov, obstajajo omejitve glede podatkov, ki jih format lahko sprejme. Te so:

- največje celo število je lahko (2^64)-1, najmanjše pa –(2^63),
- največja velikost niza je lahko (2^32)-1,
- največje število elementov v tabeli ter slovarju je lahko (2^32)-1.

Primer uporabe MessagePacka v jeziku Java je v prilogi (koda 21).

3.6 ASN.1

ASN.1 (Abstract Syntax Notation One) je standardni protokol ali jezik, ki opisuje pravila in strukture za prikaz, kodiranje, prenos ter dekodiranje podatkov v omrežjih [15]. Pomembna lastnost protokola je formalnost zapisa, ki omogoča predstavitev objektov, ki so neodvisni od tehnik kodiranj strojne opreme. Za napravo je potrebno le vedeti, kako povedati, kar želi povedati. ASN.1 torej ne upošteva nobenih posebnih standardov, načinov kodiranja programskih jezikov ali strojne opreme.

ASN.1 je skupni standard mednarodne organizacije za standardizacijo ISO, mednarodne komisije za elektrotehniko IEC ter mednarodne telekomunikacijske zveze ITU-T. Opredeljen je bil leta 1984, najnovejša različica pa 2008. Za slednjo velja, da je kompatibilna s starejšimi verzijami.

Pomembno je omeniti, da ASN.1 ni format, kot ostali. Namenjen je za specifikacijo oblike podatkov.

Standard je v vsakdanjem življenju zelo razširjen. Uporabljamo ga pri telefoniranju, uporabi bankomata, upravljanju računalniškega omrežja, kupovanju po internetu, pošiljanju elektronske pošte ter drugje.

Prednosti ASN.1:

- kompaktnost serializiranih podatkov,
- hitrost,
- minimizacija
- stabilnost.

ASN.1 definira naslednje vrste podatkov:

- tip,
- modul in obliko le-tega,
- celo število,
- logična vrednost,
- strukturni tip,
- ključne besede (npr. begin, end, import, export,…),
- oznako tipa, da se ga lahko pravilno kodira.

Pravila kodiranja

Ko programer opiše obliko in vrsto podatkov in ustvari podatke, se mora odločiti, kakšno kodiranje bo uporabil na teh podatkov. Standardna pravila kodiranja so:

- BER (basic encoding rules - osnovna pravila kodiranja),
- CER (canonical encoding rules - kanonična pravila kodiranja),
- DER (distinguished encoding rules – rahlo drugačno BER pravilo)
- XML encoding rules (XER, kodiranje v formatu XML),
- CXER (canonical XML encoding rules – rahlo drugačno XML pravilo),
- E-XER (extended XML encoding rules – rahlo drugačen XML pravilo),
- PER (packed encoding rules – paketna pravila kodiranja),
- GSER (generic string encoding rules – »generičen niz« pravilo kodiranja)

Primer uporabe standarda ASN.1

V naslednjem primeru ustvarimo nov modul z imenom ValentinovProtocol, v katerem definiramo sporočilo Oseba.

Koda 8: Definicija protokola ValentinovProtocol

```
1.  ValentinovProtocol DEFINITIONS ::= BEGIN
2.       Oseba ::= SEQUENCE {
3.            ime IA5String,
4.            starost INTEGER,
5.            seznam SEQUENCE OF Element,
6.            slovar Slovar
7.       }
8.       Element ::= IA5String
9.       Slovar ::= SEQUENCE OF ElementSlovar
10.      ElementSlovar ::= SEQUENCE {
11.           kljuc VisibleString,
12.           besedilo IA5String
13.      }
14. END
```

V protokolu definiramo sporčilo Oseba (koda 8), ki vsebuje vrednosti ime, starost, seznam in slovar. Ime je tipa niz, starost je tipa število, seznam je zbirka elementov Element, slovar pa je tipa Slovar. Element vsebuje en niz, Slovar pa vsebuje zbirko elementov ElementSlovar. Slednji vsebuje kljuc tipa niz in besedilo tipa niz.

V kodi 8 smo torej definirali osebo. V kodi 9 ustvarimo objekt »avtor«, katerega napolnimo s enakimi podatki kot v primerih pri ostalih formatih.

Koda 9: Ustvarjeno sporočilo za protokol ValentinovProtokol.

```
1.  avtor Oseba ::= {
2.      ime "Valentin",
3.      starost 23,
4.      seznam {"FRI", "VSŠ"},
5.      slovar {{kljuc "ulica", besedilo "Pod lazami"},
    {kljuc "stevilka", besedilo "32"}}
6.  }
```

Da pošljemo sporočilo preko omrežja, ga moramo najprej zakodirati. Standardna pravila kodiranja so našteta zgoraj, v naslednji kodi pa bomo zgornje sporočilo zakodirali z XML pravilom kodiranja. Zakodirano sporočilo je še vedno berljivo človeku, saj je XML tekstoven tip format.

Koda 9: Kodirano sporočilo po pravilih kodiranja XML.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <Oseba>
3.    <ime> Valentin </ime>
4.    <starost>23</starost>
5.    <seznam>
6.      <Element>FRI</Element>
7.      <Element>VSŠ</Element>
8.    </seznam>
9.    <slovar>
10.     <ElementSlovar>
11.       <kljuc>ulica</kljuc>
12.       <besedilo>Pod Lazami</besedilo>
13.     </ElementSlovar>
14.     <ElementSlovar>
15.       <kljuc>stevilka</kljuc>
16.       <besedilo>32</besedilo>
17.     </ElementSlovar>
18.   </slovar>
19. </Oseba>
```

Tako zakodirano sporočilo lahko nato pošljemo preko omrežja v drug računalnik, kjer ga preberemo z uporabo ASN.1 pravil XER dekodiranja.

3.7 Bencode

Bencode je format za kodiranje datotek za izmenjavo preko peer-to-peer sistema. Uporablja se ga za shranjevaje ter prenos strukturiranih podatkov. Večinoma se ga uporablja pri torrent datotekah. Format podpira štiri tipe podatkov [16], opisane v nadaljevanju.

Niz

Nizi so kodirani po principu (dolžina niza):(niz). Primer:

```
7:diploma
```

predstavlja niz »diploma«. Dolžina niza je zapisana v desetiški vrednosti.

Število

Števila so kodirana po principu i(število)e. Primer predstavlja število 7.

```
i7e
```

Za negativno število preprosto dodamo minus: `i-7e`. Zapis `i07e` ni pravilen, `i0e` pa.

Tudi tu je število je zapisano v desetiški vrednosti. Maksimalna in minimalna vrednost ni določena, a format podpira najmanj 64bitna števila.

Seznam

V seznam vstavimo ostale opisane tipe podatkov, tj. nize, števila, slovarje in sezname. Princip zapisa je l(kodirani tipi)e. Primer:

```
l7:diplomai7ee
```

predstavlja niz »diploma« ter število 7. Torej {»diploma«, 7}.

Primer uporabe seznama v seznamu je sledeč:

```
li7:diplomaei8:valentinee
```

kar se lepše zapiše kot {{»diploma«}{»valentin«}}.

Slovar

Slovar se kodira na sledeč način: d(kodiran tip)(kodiran element)e. Primer:

```
d7:diplomai10ee
```

predstavlja slovar {»diploma« -> 10}.

Zapis primera iz kode 1 v Bencode formatu je sledeč:

```
l8:Valentini23el3:FRI3:VSSed5:ulica10:Pod
Lazami8:stevilkai32eee
```

Bencode je enostaven, fleksibilen ter se z načinom kodiranja števil ter besedila izogne problemom pri branju serializiranih podatkov na različnih sistemih, kar je pomembno za aplikacije kot so BitTorrent.

3.8 Smile

Smile je binarni format za izmenjavo podatkov, ki bazira na formatu JSON. V primerjavi s slednjim je Smile kompaktnejši ter bolj učinkovit za procesiranje (pisanje ter branje podatkov) [18]. To doseže z binarnim kodiranjem (podobno kot pri formatu BSON) ter

uporabo povratnih referenc (back reference), ki omogočajo zamenjavo oznak (ki so velikosti do 64 bajtov) z referenčnimi IDji (ki so velikosti 1 ali 2 bajta).

Trenutno znane knjižnice, ki podpirajo Smile format, so napisane za Javo ter C/C++.

Lastnosti formata:

- nizi so kodirani po sistemu kodiranja UTF-8,
- števila so lahko velikost do 64 bitov,
- optimizacija zapisa enakih nizov. Ko razčlenjevalnik generira Smile datoteko, lahko vsakič, ko naleti na niz, preveri, ali se je ta niz že pojavil. V tem primeru zapiše le referenčno število, ki kaže na naslov prvotnega enakega niza,
- enako optimizacijo ima format tudi za imena objektov.

Implementacija v jeziku Java

Smile se v Javi uporablja na enak način kot JSON, le da se namesto JsonFactory (tj. glavni razred za podporo JSONu v Javi) uporabi SmileFactory (tj. glavni razred za podporo formata Smile v Javi).

Koda 10: Primer uporabe formata Smile v jeziku Java.

```
1.  public class SmilePrimer{
2.      public static void main(String[] args) throws
    IOException, ClassNotFoundException {
3.          List seznam= new ArrayList();
4.          seznam.add("FRI");
5.          seznam.add("VSŠ");
6.          Dictionary slovar= new Hashtable();
7.          slovar.put("ulica", "Pod Lazami");
8.          slovar.put("stevilka", "32");
9.          Oseba oseba= new Oseba("Valentin", 23, seznam,
                slovar);
10.
11.         SmileFactory f= new SmileFactory();
12.         ObjectMapper mapper= new ObjectMapper(f);
13.         byte[] serializirano=
                mapper.writeValueAsBytes(oseba);
14.         Oseba oseba2 = mapper.readValue(serializirano,
                Oseba.class);
15.     }
16.
17.     static class Oseba implements Serializable{
18.         private String ime;
19.         private int starost;
20.         private List seznam;
21.         private Dictionary slovar;
22.
23.         public Oseba(String ime, int starost, List
              seznam, Dictionary slovar) {
24.             this.ime = ime;
25.             this.starost = starost;
26.             this.seznam = seznam;
27.             this.slovar = slovar;
28.         }
29.     }
30. }
```

V vrsticah tri do deset (koda 10) ustvarimo objekt tipa Oseba. Nato v vrsticah enajst in dvanajst ustvarimo objekt za delo s formatom Smile. V trinajsti vrstici serializiramo osebo »oseba« v tok podatkov. Tega bi lahko zapisali v datoteko ali prenesli preko interneta v drug sistem. V štirinajsti vrstici ustvarimo nov objekt tipa Oseba, katerega vrednosti so deserializiran tok podatkov, ki je bil ustvarjen v trinajsti vrstici.

3.9 Protocol Buffers

Protocol Buffers je format ali metoda za serializacijo strukturiranih podatkov. Vključuje vmesniški jezik za definiranje strukture podatkov ter prevajalnik. Namen uporabe Protocol Buffers je olajšati omrežno komunikacijo, njeno enostavnost in hitrost [17].

Protocol Buffers je ustvarilo podjetje Google za interno uporabo v podjetju, pri čemer je tudi ustvarilo prevajalnik, imenovan protoc, za programske jezike Javo, C++ in Python, ki imajo odprtokodno licenco. Obstajajo pa tudi prevajalniki za ostale programske jezike. Prevajalnik se uporablja za generiranje kode v določenem programskem jeziku (katerem je odvisno od prevajalnika). S pomočjo te kode lahko nato ustvarimo in beremo objekte v izbranem programskem jeziku.

Proces uporabe Protocol Buffers je sledeč:

- uporabnik ustvari datoteko (imenovano proto datoteka), v kateri definira podatkovne strukture, imenovane sporočila (ang. messages),
- uporabnik uporabi prevajalnik, ki generira razrede v izbranem programskem jeziku. Ti razredi predstavljajo strukturo sporočil, ki jih je uporabnik definiral v proto datoteki, vsebujejo pa metode za dostop in spreminjanje vrednosti polj omenjenih sporočil.

Vsako sporočilo ima polja, ki so lahko:

- število (celo ali decimalno število),
- logična vrednost (bool),
- besedilo (string),
- drugo sporočilo.

Vsako polje pa je lahko:

- obvezno (required),
- neobvezno (optional),
- ponovljivo (repeated).

Če je polje obvezno, ga moramo vedno inicializirati, če je neobvezno, ga seveda ni potrebno inicializirati. V tem primeru bo polje dobilo privzeto vrednost, katero določimo v proto datoteki. Če je polje ponovljivo, se lahko v sporočilu poljubnokrat ponovi (tudi ničkrat),

Primer uporabe formata

V naslednjem primeru bomo ustvarili proto datoteko, jo prevedli z uporabo prevajalnika ter na koncu prikazali kodo, ki jo je prevajalnik generiral. Uporabljen bo prevajalnik za Javo, kar pomeni, da nam bo le-ta ustvaril kodo za delo v programskem jeziku Java.

1. Kreiranje proto datoteke.

Koda 11: Proto datoteka za primer Oseba v formatu Protocol Buffers.

```
1.   package PBprimer;
2.   option java_package = "com.example.PBprimer";
3.   option java_outer_classname = "PBOseba";
4.
5.   message Oseba {
6.     required string ime = 1;
7.     required int32 starost = 2;
8.     repeated string seznam = 4;
9.     required Slovar slovar = 5;
10.
11.    message Slovar {
12.      required string kljuc = 1;
13.      required string besedilo = 2;
14.    }
15.  }
```

V prvi vrstici kode 11 deklariramo paket, v katerem bo aplikacija, ker nam je to v pomoč, ko imamo več delov pri projektu. Z uporabo opcije »java_package« v drugi vrstici določimo kateri Java paket bo vseboval razrede v nadaljevanju, z opcijo »java_outer_classname« pa definiramo ime razreda, ki bo vseboval vse ostale razrede, ki bodo ustvarjeni s to datoteko. Slednja opcija je neobvezna. Če je ne določimo, se bo za ime uporabilo kar ime datoteke v rahlo drugačni obliki. Primer: ime datoteke iz primera je »ProtocolBuffers primer koda.proto«. Protocol Buffers bo razrede shranil v razred »ProtocolBuffersPrimerKoda«.

V nadaljevanju definiramo sporočila Oseba in Slovar. Sporočilo Oseba ima polja ime, starost, seznam in slovar. Slovar ima polji kljuc in besedilo. Vsako polje ima tudi t.i. oznako s številko (required string ime = ***1***). Te služijo zgolj učinkovitejšemu binarnemu kodiranju. Pomembno je le vedeti, da je zaželena uporaba števil med 1 in 15, spet, zaradi učinkovitejšega kodiranja. Za polja definiramo tudi tip, kot so niz (string) za ime osebe in int (število) za starost osebe.

Vidimo torej, da bomo lahko v Javi ustvarili osebo, kateri moramo nujno definirati ime in starost, seznam in slovar.

Ustvarjeno proto datoteko shranimo kot »PBOseba.proto«.

2. Uporaba prevajalnika

Da lahko prevedemo proto datoteko, moramo najprej namestiti prevajalnik, katerega dobimo na uradni strani Protocol Buffers. Nato v ukazno vrstico vpišemo naslednji ukaz:

```
protoc –I=[1] –java_out=[2] [3]/PBOseba.proto
```

Pri čemer pa:

1. »[1]« zamenjamo s potjo do mape na disku, kjer imamo kodo aplikacije,
2. »[2]« zamenjamo s potjo, kamor želimo, da se bo shranila generirana koda (navadno je to enak pot kot pri prvi točki),
3. »[3]« zamenjamo s potjo, kjer se nahaja naša proto datoteka.

Poglejmo generirane datoteke, ki jih je ustvaril prevajalnik. Opazimo, da je le-ta ustvaril datoteko »PBOseba.java«. V njej je definiran razred PBOseba, v katerem so definirani razredi za sporočilo Oseba in Slovar, ki smo jih definirali v proto datoteki. Vsak razred ima tudi svojega graditelja (ang. Builder). Builder je razred, s katerim ustvarimo objekte tipov, katere smo definirali v proto datoteki (v zgornjem primeru sta ta tipa Oseba in Slovar). Ko ustvarimo tak objekt, ga lahko napolnimo z informacijami (dodamo ime, starost, seznam in slovar) ter na koncu kličemo nad objektom metodo `build()` razreda Builder, s čimer končamo delo na tem objektu. V naslednji kodi je glavni del razreda Oseba, ki ga je generiral prevajalnik.

Koda 12: Generiran razred sporočila Oseba.

```
1.  public boolean hasIme();
2.  public String getIme();
3.
4.  public boolean hasStarost();
5.  public int getStarost();
6.
7.  public List<Slovar> getSlovarList();
8.  public int getSlovarCount();
9.  public Slovar getSlovar(int index);
10.
11. public List<String> getSeznamList();
12. public int getSeznamCount();
13. public String getSeznam(int index);
```

Opazimo, da ima vsako polje svoje metode (koda 12). Polje »Ime« ima metodi `hasIme()` in `getIme()`, s katerimi dostopamo do podatkov o imenu, polje »Slovar« pa `getSlovarList()` za dostop do seznama slovarjev, `getSlovarCount()` za dostop do števila slovarjev ter `getSlovar(int index)` za dostop do slovarja na določenem mestu v seznamu slovarjev.

Poglejmo si še razred Builder sporočila Oseba.

Koda 13: Generiran Builder razred sporočila Oseba.

```
1.  public boolean hasIme();
2.  public String getIme();
3.  public Builder setIme(String value);
4.  public Builder clearIme();
5.
6.  public boolean hasStarost();
7.  public int getStarost();
8.  public Builder setStarost(int balue);
9.  public Builder clearStarost();
10.
11. public List<Slovar> getSlovarList();
12. public int getSlovarCount();
13. public Slovar getSlovar(int index);
14. public Builder setSlovar(int index, Slovar value);
15. public Builder addSlovar(Slovar value);
16. public Builder addAllSlovar(Iterable<Slovar> value);
17. public builder clearSlovar();
18.
19. public List<String> getSeznamList();
20. public int getSeznamCount();
21. public String getSeznam(int index);
22. public Builder setSeznam(int index, Seznam value);
23. public Builder addSeznam(Seznam value);
24. public Builder addAllSeznam(Iterable<String> value);
25. public builder clearSeznam();
```

Opazimo, da ima vsako polje še dodatne metode za dodajanje oziroma spreminjanje vrednosti polja, kot so `setIme(String value)` za navedbo imena ter `clearIme()` za izbris vrednosti imena (koda 13).

Nasveti pri daljši uporabi Protocol Buffers

Če želimo, da je naš Protocol Buffers za nazaj in naprej kompatibilen, moramo slediti naslednjim nasvetom:

1. ne spreminjati oznak polj v sporočilih,
2. ne dodajati ali brisati polj, ki smo jih označili kot obvezna (required),
3. dovoljeno je brisati neobvezna ali ponovljiva polja,
4. dovoljeno je dodajati nova neobvezna ali ponovljiva polja, če za njih uporabimo nove oznak,.
5. priporočljivo je, da se polj ne označi kot obvezna, ampak kot neobvezna ali ponovljiva.

3.10 Serializacija v programskem jeziku Java

Programski jezik Java vsebuje vgrajeno podporo za serializacijo objektov v tok podatkov ter deserializacijo toka podatkov v objekt. Objekt oziroma razred, ki ga želimo serializirati, mora implementirati vmesnik Serializable. Vmesnik ne vsebuje nobenih metod. Važen je zgolj za to, da sporoči prevajalniku Jave naj vključi mehanizem za serializacijo tega objekta. Če za neko atribut v objektu nočemo da se serializira, definiramo to polje kot »static« ali »transient«.

Za serializacijo objekta v datoteko se v programskem jeziku Java uporablja razreda ObjectOutputStream() skupaj z FileOutputStream(). ObjectOutputStream() konvertira java objekt v tok podatkov ter shrani v OutputStream() (ki je lahko FileOutputStream(), ByteArrayOutputStream(), itd.). FileOutputStream sprejme tok podatkov in ga shrani v datoteko.

Za deserializacijo objekta v datoteko se v javi uporablja ObjectInputStream() skupaj z FileInputStream(). Slednji prebere datoteko ter ustvari tok podatkov.

ObjectInputStream() ustvari java objekt iz toka podatkov.

Primer uporabe serializacije ter deserializacije

V okolju NetBeans prikažimo primer serializacije ter nato še deserializacije.

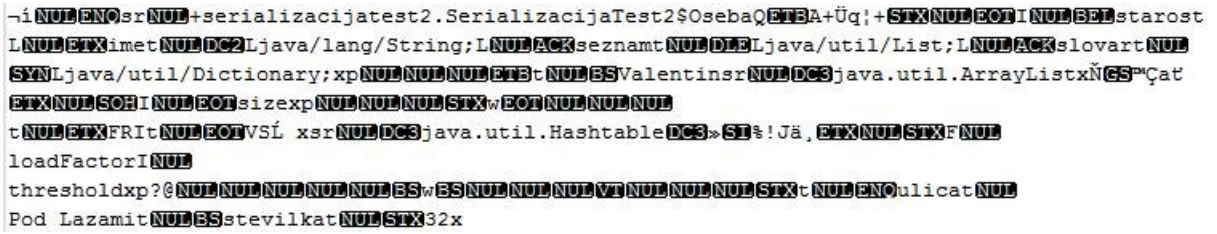
a. Serializacija

Koda 14: Serializacija primera v Javi

```
1.   public class XMLPrimer{
2.       public static void main(String[] args) {
3.           List seznam= new ArrayList();
4.           seznam.add("FRI");
5.           seznam.add("VSŠ");
6.           Dictionary slovar= new Hashtable();
7.           slovar.put("ulica","Pod Lazami");
8.           slovar.put("stevilka","32");
9.
10.          Oseba oseba = new Oseba("Valentin", 23, seznam,
             slovar);
11.          FileOutputStream fos= new
                 FileOutputStream("oseba.txt");
12.          ObjectOutputStream oos= new
                 ObjectOutputStream(fos);
13.          oos.writeObject(oseba);
14.      }
15.
16.      static class Oseba implements Serializable{
17.          private String ime;
18.          private int starost;
19.          private List seznam;
20.          private Dictionary slovar;
21.
22.          public Oseba(String ime, int starost, List
                 seznam, Dictionary slovar) {
23.              this.ime = ime;
24.              this.starost = starost;
25.              this.seznam = seznam;
26.              this.slovar = slovar;
27.          }
28.      }
29.  }
```

V kodi 14 je predstavljena serializacija objekta Oseba. Od tretje do 9 vrstice ustvarimo seznam in slovar ter ju oba napolnimo s podatki. Nato v deseti ustvarimo novo osebo, katero s pomočjo razredov `ObjectOutputStream` ter `FileOutputStream` v trinajsti vrstici serializiramo v datoteko »oseba.txt«.

Če datoteko »oseba.txt« odpremo, vidimo sledeče.



Slika 3.1: Serializirani podatki, ustvarjeni v jeziku Java.

Besedilo v datoteki ni berljivo za ljudi, ker so podatki binarizirani. Važno je le, da ga zna program prebrati ter poustvariti objekt.

b. Deserializacija

Koda 15: Deserializacija primera v Javi.

```
1.  public class XMLPrimer{
2.      public static void main(String[] args) throws
    IOException, ClassNotFoundException {
3.          FileInputStream fis= new
                FileInputStream("oseba.txt");
4.          ObjectInputStream ois= new
                ObjectInputStream(fis);
5.          Oseba oseba2 = (Oseba) ois.readObject();
6.      }
7.
8.      static class Oseba implements Serializable{
9.          private String ime;
10.         private int starost;
11.         private List seznam;
12.         private Dictionary slovar;
13.
14.         public Oseba(String ime, int starost, List
    seznam, Dictionary slovar) {
15.             this.ime = ime;
16.             this.starost = starost;
17.             this.seznam = seznam;
18.             this.slovar = slovar;
19.         }
20.     }
21. }
```

V tretji in četrti vrstici ustvarimo objekta FileInputStream ter ObjectInputStream, s pomočjo katerih nato v peti vrstici deserializiramo podatke iz datoteke »oseba.txt« v objekt oseba2.

Poglavje 4

Primerjava formatov

Po pregledu in opisu popularnejših formatov za serializacijo podatkov je pomembno vedeti, kateri so uporabni za določeno situacijo. Formati so si namreč različni po hitrosti, učinkovitosti, berljivosti podatkov, kompaktnosti ter drugih lastnostih. Za grobo primerjavo preglejmo pomembnejše značilnosti vseh opisanih formatov.

Format	**Bazira na**	**Standarden**	**Binaren / tekstoven**	**shema**	**Podpora v jezikih**
XML	SGML	da	tekstoven	lahko	praktično vsi
JSON	JavaScript	da	tekstoven	delno	praktično vsi
BSON	JSON	da	binaren	ne	C, C#, C++, Delphi, Java, Lisp, Perl, PHP, Python, Ruby, …
YAML	XML, JSON, SAX, HTML, Java, Python,…	da	tekstoven	delno	C, C++, Ruby, Python, Java, Perl, C3, PHP, Javascript, .NET
MessagePack	grobo JSON	da	binaren	ne	Java, Python, Ruby, C#, Rust, C++, C, PHP, Go, Delphi
ASN.1	/	da	binaren	da	Java, Python, C, C++
Bencode	/	da	binaren	ne	Java, Python, C, Perl
Smile	JSON	da	binaren	ne	Java, C, C++
Protocol Buffers	/	delno	binaren	da	Java, C, Python, C++, C#, Delphi, Javascript, PHP, Ruby, Python, …

Tabela 4.1: Splošna primerjava formatov [3].

Za lažjo primerjavo razdelimo formate glede na to, ali so binarni ali tekstovni.

Binarni formati:

- ASN.1
- BSON
- Java serialization
- MessagePack
- Protocol Buffers
- Smile
- Bencode

Tekstovni formati:

- JSON
- XML
- YAML

Sprva se moramo odločiti, ali nam bolj ustreza binarni ali tekstovni format. V nadaljevanju so opisane prednosti in slabosti obeh vrst formatov.

Prednosti in slabosti tekstovnega formata

Tekstovni format je človeku razumljivejši že na pogled. To je pomembno, ko želimo, da tudi drugi ljudje lažje razumejo s kakšnimi podatki delamo. Pomembna prednost je tudi možnost lažja prenosljivosti serializiranih podatkov v drugo delovno okolje.

Glavna slabost tekstovnih formatov je poraba prostora, ki je večja kot bi bila v binarni obliki. Temu sledi še ena slabost, da se za serializacijo in deserializacijo porabi več časa.

Prednosti in slabosti binarnega formata

Za binarno serializirane podatke je predvsem značilno, da porabijo manj prostora kot tekstovno predstavljeni podatki. Temu sledi prednost, da se podatke hitreje serializira in deserializira, saj so že v računalniku razumljivi obliki. Uporaba binarnega formata je primerna, ko vemo, da bodo aplikacije temeljile na enakem sistemu, recimo v programskem jeziku Java. Za večino binarnih formatov je značilno, da jih ni težko implementirati, kar prihrani čas.

Pomembna slabost binarnih formatov je, da niso nujno boljši od tekstovnih formatov v smislu hitrosti delovanja in kompaktnosti podatkov. Poleg tega je težje poiskati rešitev, če pride do morebitnih težav pri manipulaciji podatkov, ker serializiranih podatkov ne moremo prebrati. Pomembno je, da se vsak programer ustrezno odloči, kakšen format je najboljši za njegovo situacijo.

4.1 Primerjava tekstovnih formatov

4.1.1 XML in JSON

Tako XML kot JSON lahko uporabljamo za podobne namene. Glavne razlike vidimo v naslednji tabeli.

Lastnost	**XML**	**JSON in YAML**
Podpora podatkovnih tipom	Ne	zagotavlja skalarne podatkovne tipe* ter strukturirane podatke z uporabo tabel in objektov
Podpora tabelam	ne podpira, tabele se predstavi z gnezdenjem podatkov	da
Podpora slovarju	ne, navadno se uporablja pare atributov in elementov.	da
Podpora »null« vrednosti	potrebno je uporabiti xsi:nil v elementu in uvoziti ustrezni imenski prostor	da.
Podpora komentarjem	da	ne
Podpora imenskim prostorom	da	ne, navadno se v imenu uporabi predpono
Podpora oblikovanju podatkovne strukture	ne, zahteva trud, da določimo, kako bomo aplikacijske podatke preslikali v XML obliko	lažje, zagotavlja bolj neposredno preslikavo aplikacijskih podatkov
Velikost	relativno velika datoteka	kompaktna sintaksa, manjša velikost
Zahtevnost uporabe	navadno zahteva uporabo drugih tehnologij, na primer DOMa	enostavno za uporabo

Tabela 4.2: Primerjava lastnosti med XML in JSON [8].

*skalarni podatkovni tipi so na primer celo in decimalno število, besedilo ter logična vrednost.

Iz zgornje tabele lahko sklepamo, da se JSON uporablja, ko imamo opravka s strukturiranimi podatki, kot so tabele in slovarji, torej s podatki, katere se uporablja v programskih jezikih. V praksi je XML najbolje uporabljati za markiranje dokumentov, JSON pa, ko se uporablja programske jezike, kot so Java, C++ in Python.

4.1.2 YAML in JSON

YAML je po zgradbi zelo podoben JSONu. Oba predstavita podatke v tekstovni obliki ter imata podobno sintakso. Razlika je v ciljih formatov. Cilj JSONa je enostavnost in univerzalnost, zato je lahko generirati ter razčlenjevati JSON datoteke. Cilj YAMLja pa je, da se podatke predstavi v človeku čim razumljivejši sintaksi ter boljša podpora podatkovnih strukturam. Posledica je kompleksnejše generiranje ter razčlenjevanje kot pri formatu JSON ter manjša univerzalnost oziroma prenosljivost med različnimi programskimi okolji.

Tehnično gledano je format YAML nadgradnja JSONa. Vsebuje vse kar ima JSON in dodatne nadgradnje, kot so na primer pisanje komentarjev, ponavljanje določenih delov, pisanje nizov brez uporabe narekovajev ter drugo. Poleg tega velja, da se vsako JSON datoteko lahko brez težav prenese v YAML format.

V praksi se JSON večinoma uporablja v spletnih tehnologijah (na primer v AJAXu (tj. skupek tehnologij za asinhrone spletne aplikacije)), medtem ko se YAML več uporablja pri podatkovnih procesih, pri katerih se ne uporablja omrežja.

4.2 Primerjava binarnih formatov

4.2.1 BSON, Smile, MessagePack in Bencode

Tako Smile kot BSON kot MessagePack so formati, ki bazirajo na JSONu, katerega dopolnijo z binariziranjem podatkov. Poleg tega ima vsak format še dodatne funkcije z namenom učinkovitejšega razčlenjevanja podatkov (razložene v opisih formatov). Seznamu dodajam še format Bencode, ki je tako kot ostali binaren format brez sheme podatkov.

Najprej primerjajmo BSON in MessagePack.

Kompatibilnost s formatom JSON: presenetljivo je MessagePack bolj kompatibilen z JSONom kot pa BSON. Glaven razlog je, ker je MessagePack med drugim tudi namenjen temu, da se brez težav pretvori v ali iz JSON oblike. Razlog, zakaj BSON ni bolj kompatibilen, je, ker lahko vsebuje posebne tipe podatkov, ki jih JSON ne podpira, kar pomeni, da se pri pretvorbi podatkov iz BSON v JSON le ti izgubijo.

Velikost: Za enake podatke bo MessagePack datoteka manjša od BSON datoteke, kar pomeni, da je končna velikost serializiranih podatkov manjša.

Hitrost: MessagePack je hitrejši od BSONa v serializaciji in deserializaciji. Pri serializaicji je tudi pomembno, kakšne podatke serializiramo. BSON se namreč odlično izkaže pri serializaciji slik in filmov.

Dinamično spreminjanje serializiranih podatkov: vzemimo za primer slovar »{»Valentin«:1, »Kragelj«:2}«. V njem želimo vrednost 1 spremeniti na 1000. V MessagePacku se vrednost 1 zapiše v 1 bajt, vrednost 1000 pa v 3 bajte. To pomeni, da moramo drugi element slovarja (»Kragelj«:2) premakniti za 2 bajta, da lahko zamenjamo vrednost 1 v 1000. V formatu BSON tako vrednost 1 kot 1000 potrebujeta 5 bajtov, kar pomeni, da pri zamenjavi vrednosti ne potrebujemo premikati ostalih elementov v slovarju. Sklepamo torej, da je format BSON boljši v primeru, ko moramo dinamično spreminjati podatke.

Podpora RPCju: RPC (remote procedure call) je proces, ki omogoča programu, da izvede določen proces ali podprogram v drugem računalniku, ki je na istem omrežju. Format MessagePack slednje podpira, medtem kot BSON ne podpira.

Iz zgornjih dejstev lahko zaključimo, da je format MessagePack v večini primerov primernejši za implementacijo kot BSON. Tudi v praksi uživa MessagePack več podpore.

Kot zanimivost se tu lahko omeni spletna storitev na domači spletni strani formata MessagePack [14]. Tam se lahko v realnem času primerja velikost podakotv v MessagePack in JSON formatu.

Format Smile do sedaj še ni bil podrobneje omenjen. Razlog za to je, ker se o njem v praksi ne veliko govori, kar vpliva na njegovo popularnost in podporo v programskih jezikih. Po učinkovitosti je zelo podoben formatu BSON, le da poskuša izboljšati serializacijo in deserializacijo z drugimi funkcijami (opisanimi v opisu formata).

Preden se odločimo, katerega od zgornjih formatov uporabiti, je potrebno vedeti, da noben od le-teh ni primeren, če v našem projektu še ne uporabljamo formata JSON. Če ga, pa najprej preverimo, ali nam format MessagePack ustreza glede na zgoraj opisana dejstva. Če nam ne, se obrnemo na formata BSON ali Smile. Katerega uporabimo, je odvisno od tega, katere funkcije od obeh formatov mislimo, da bi nam bolje koristile. Razen slednjega je izbira formata odvisna zgolj od programerjeve osebne preference.

Če formata JSON ne uporabljamo, je Bencode dobra izbira. Čeprav za njega ne obstaja veliko knjižnic za programske jezike in se ga v praksi bolj uporablja v peer-to-peer programih (kot so BitTorrent), se format po hitrosti serializiranja in kompaktnosti podatkov lahko primerja z ostalimi formati.

4.2.2 ASN.1 in Protocol Buffers

Tako za ASN.1 kot za Protocol Buffers je značilno, da je potrebno sprva ustvariti shemo, kjer definiramo podatkovne strukture podatkov, ki jih bomo prenašali, ter uporabimo prevajalnik

za generiranje potrebne kode. Slednje je dobra iztočnica za primerjavo teh dveh formatov, saj sta edina od opisanih, ki potrebujeta shemo.

Glede hitrosti ter kompaktnosti podatkov sta si formata zelo blizu. Velikost serializiranih podatkov je predvsem odvisna od tega, kakšne tipe podatkov shranjujemo, hitrost formata pa od tega, kateri tip kodiranja izberemo. Protocol Buffers uporablja podjetje Google za večino interne komunikacije, kar kaže na stabilnost ter praktičnost uporabe formata. Slaba stran Protocol Buffers formata je podpora programskim jezikom. Razen knjižnic za jezike C++, Python in Java (katere je Google sam ustvaril), podpora drugim jezikom ter kvaliteta podpore ni nujno zadovoljiva. Slednjemu botruje tudi dejstvo, da format ni sprejet kot standard. ASN.1 po drugi strani je standard, zaradi česar je širše uporabljen. Dobra lastnost formata je tudi, da uporabnik lahko izbere tip kodiranja. Za hitro, binarno kodiranje je najboljša izbira DER kodiranje, za tekstovno kodiranje, združljivo s formatom XML, pa XER kodiranje. Slabe lastnosti so predvsem malo knjižnic za podporo formatu (za programska jezika C# in Java je ena redkih kvalitetnih knjižnic »Bouncy Castle«), kompleksnost popolne implementacije formata, nepopularnost (kljub razširjenosti) ter namen formata. ASN.1 se namreč uporablja za definiranje abstraktnih podatkov in specifikacijo pravil za kodiranje in ne toliko za kodiranje podatkovnih tipov in objektov v programiranju.

V realnem svetu se format Protocol Buffers večinoma uporablja v interni prenosljivosti podatkov v podjetjih, medtem ko se ASN.1 uporablja na področju telekomunikacij, na primer v mobilnih telefonih, letalih, satelitih, bankomatih ipd.

4.2.3 Primerjava formatov in podpore serializaciji v programskih jezikih

Omenili smo, da obstajajo objektno usmerjeni programski jeziki, ki imajo lastno podporo za serializacijo. Za slednje bomo kot primer vzeli podporo serializaciji v jeziku Java.

V opisu serializacije v Javi je prikazan primer uporabe, sedaj pa opišimo dobre in slabe strani uporabe internih rešitev za serializacijo v programskih jezikih.

Dobre lastnosti:

- enostavna implementacija in uporaba,
- enostavno delo s kompleksnimi podatki oziroma objekti.

Slabe lastnosti:

- ni priporočljivo spreminjati vsebine razreda po tem, ko smo ga že serializirali,
- vsak razred mora biti vnaprej definiran na ciljnem računalniku (da se lahko tok podatkov deserializira).

Praktično gledano je uporaba interne podpore serializaciji v programskih jezikih primerna le, ko se serializirane podatke uporablja zgolj v okolju, v katerem delamo (na primer če smo podatke serializirali v jeziku Java, je priporočljivo z njimi delati zgolj v Javi), nam velikost serializiranih podatkov in hitrost prenosa le-teh ni prioriteta ter želimo enostavno delo.

Poglavje 5

Testiranje formatov

S pomočjo programskega jezika Java smo v okoljih Eclipse in NetBeans ustvarili programe za testiranje hitrosti serializacije in deserializacije ter velikost podatkov za nekatere od opisanih formatov. Testirali smo formate XML, JSON, YAML, MessagePack, Bencode in podporo serializacije v Javi. Prve štiri formate smo izbrali, ker spadajo med popularnejše formate, Bencode ker želimo s testi dokazati, da je zelo uporaben format kljub nepopularnosti, serializacijo v Javi pa kot predstavnika programskih jezikov z interno podporo serializaciji.

Testiranje smo razdelili na tri dele.

Pri prvem smo testirali hitrost serializacije tabele velikosti 100 elementov, ki je vsebovala nize »test«, ter hitrost serializacije tabele velikosti 100 elementov, ki je vsebovala naključna števila med 0 in 2^31.

Pri drugem testu smo testirali hitrost deserializacije tabele velikosti 100 elementov, ki je vsebovala nize »test«, ter hitrost deserializacije tabele velikosti 100 elementov, ki je vsebovala naključna števila med 0 in 2^31.

Pri tretjem testu smo testirali velikost serializiranih podatkov.

Vsa testiranja so bila opravljena v operacijskem sistemu Windows 7 v programskih okoljih Eclipe ter NetBeans.

V nadaljevanju je kot primer prikazana serializacija v formatu YAML.

Koda 16: Serializacija tabele stotih nizov.

```
1.  static void testMake() throws IOException{
2.       String s= "";
3.       for (int i = 0; i < 100; i++) {
4.            s += "\n- test";
5.       }
6.
7.       Yaml yaml = new Yaml();
8.       List<String> list = (List<String>) yaml.load(s);
9.       System.out.println(list.toString());
10.
11.      FileWriter file = new FileWriter("testni.yaml");
12.      file.write(list.toString());
13.      file.flush();
14.      file.close();
15. }
```

Najprej ustvarimo seznam stotih elementov (pri čemer je vsak element niz »test«), nato ustvarimo objekt tipa YAML in vanj vnesemo tabelo ter na koncu zapišemo serializirane podatke v YAML formatu v datoteko »testni.yaml« (koda 16). Za serializacijo tabele z naključnimi števili je koda enaka, le da v zanki dodajamo namesto besedila »test« naključno število med 0 in 2^31.

Deserializacija v formatu YAML je sledeča.

Koda 17: Deserializacija tabele stotih nizov.

```
1.   static void testRead(){
2.        final Yaml yaml = new Yaml();
3.         Reader reader = null;
4.         LinkedHashMap lhm = new LinkedHashMap();
5.
6.         try {
7.             reader = new FileReader("testni.yaml");
8.             ArrayList<String> deser= (ArrayList<String>)
                   yaml.load(reader);
9.         } catch (final FileNotFoundException e) {
10.            System.err.println("Datoteka ne obstaja:"+e);
11.        } finally {
12.            if (null != reader) {
13.                try {
14.                    reader.close();
15.                } catch (final IOException e) {
16.                   System.err.println("Napaka na koncu”);
17.                }
18.            }
19.        }
20.  }
```

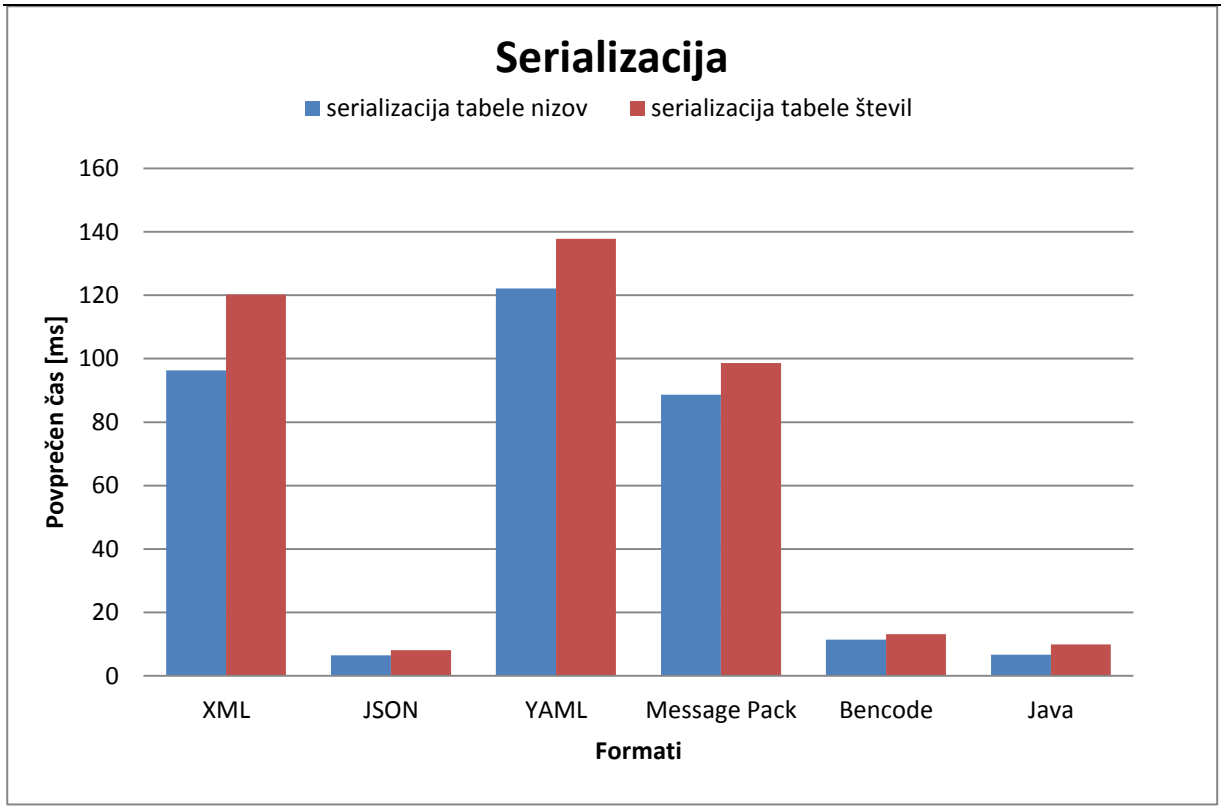
V kodi 17 naložimo datoteko »testni.yaml« v object reader ter napolnimo seznam »deser« z uporabo metode `load()`. Koda za preostale formate je priložena v prilogi diplomske naloge.

Za hitrosti serializacije tabele nizov in tabele števil smo dobili naslednje meritve.

Format	**Povprečje testiranj hitrosti serializacije tabele nizov [ms]**	**Povprečje testiranj hitrosti serializacije tabele števil [ms]**
XML	96,3	120,2
JSON	6,5	8,1
YAML	122,2	137,8
MessagePack	88,6	98,6
Bencode	11,4	13,1
Podpora v Javi	6,7	9,9

Tabela 5.1: Meritve hitrosti serializacije obeh testov.

Obe povprečji za vsak format za lažjo predstavo prikažemo v grafikonu 5.1.



Grafikon 5.1: Povprečne hitrosti serializacije tabele nizov ter tabele števil.

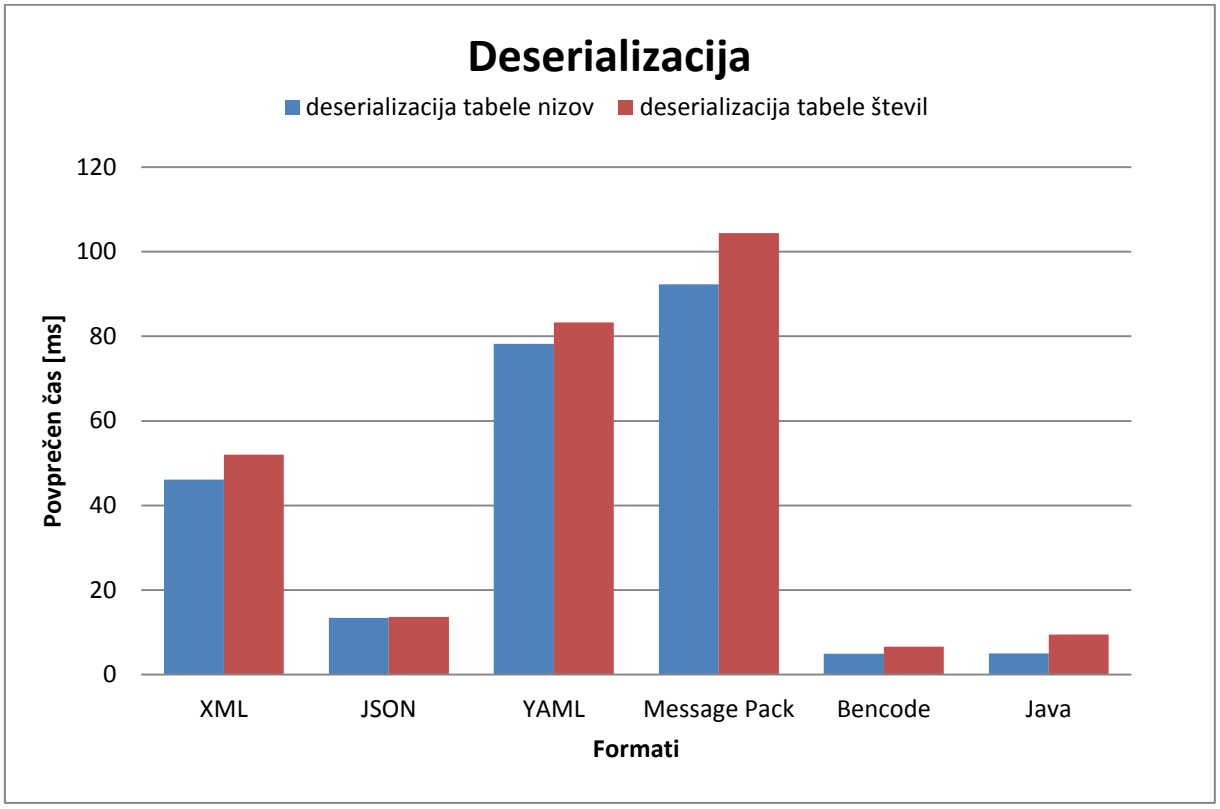
V zgornjem grafu opazimo, da so za serializacijo podatkov tekstovni formati (XML in YAML) časovno požrešnejši. Format YAML je najpočasnejši, kar je razumljivo, saj je glavni cilj formata to, da se podatke predstavi v človeku čim razumljivejši obliki. JSON je presenetljivo potreboval relativno kratek čas. Razlog za to je, ker za razliko od na primer formata XML, vsebuje JSON podporo strukturiranih podatkom, kot je v našem primeru tabela. Format MessagePack je presenetljivo potreboval več časa, kot bi predvidevali. Razlog za to sklepamo da je v nepopolnosti testov in okolja, v katerem smo poganjali teste. Opazimo tudi, da vsi formati potrebujejo več časa za serializacijo tabele števil. Razlog je v velikosti elementov; velika števila potrebujejo več prostora kot nizi dolžine štirih črk.

Za deserializacijo tabele nizov in tabele števil smo dobili naslednje meritve.

Format	**Povprečje testiranj hitrosti deserializacije tabele nizov [ms]**	**Povprečje testiranj hitrosti deserializacije tabele števil [ms]**
XML	46,1	52
JSON	13,4	13,6
YAML	78,2	83,3
MessagePack	92,3	104,4
Bencode	4,9	6,6
Podpora v Javi	5	9,4

Tabela 5.2: Meritve deserializacije obeh testov.

Obe povprečji za vsak format predstavimo še v grafikonu 5.2.



Grafikon 5.2: Povprečne hitrosti deserializacije tabele nizov ter tabele števil.

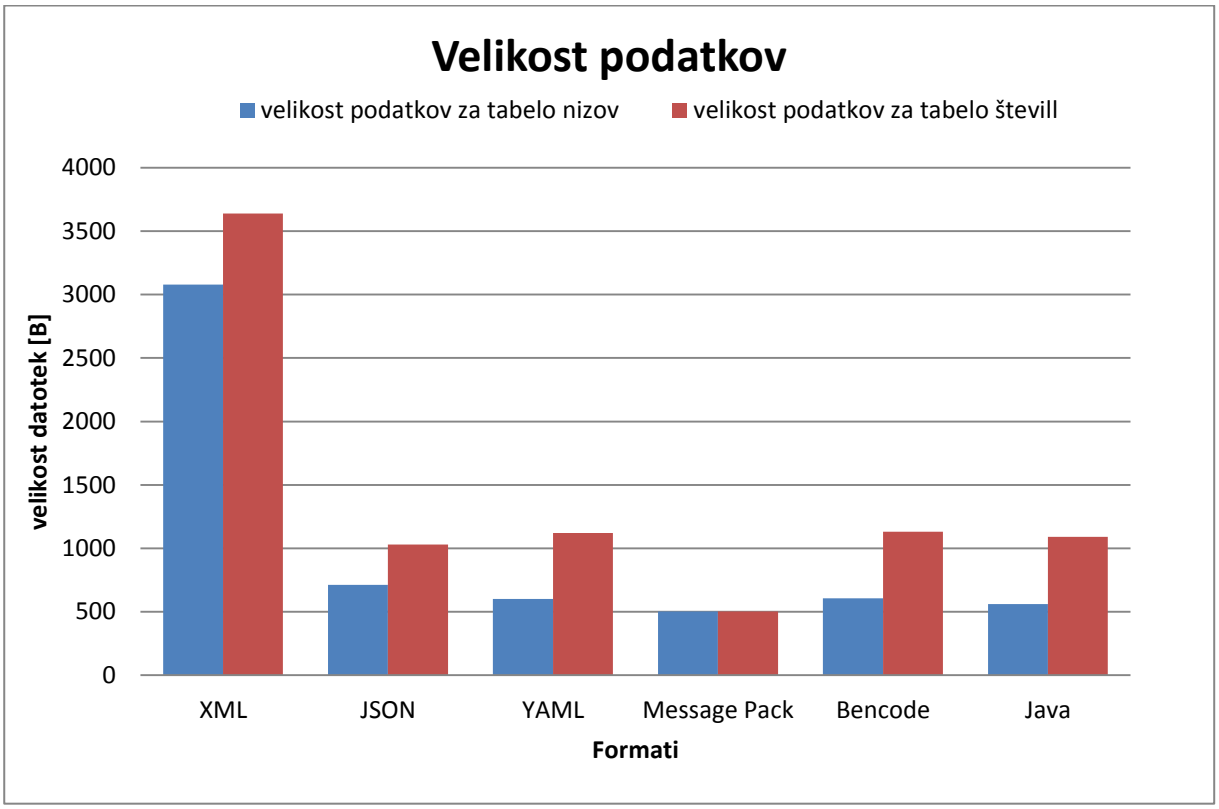
Graf hitrosti deserializiranja je zelo podoben grafu serializiranja, le da je v tem primeru format YAML hitrejši od formata MessagePack za povprečno 14 milisekund. Ponovno opazimo, da vsi formati potrebujejo več časa za serializacijo tabele števil.

Za velikost podatkov obeh serializiranih tabel (z niz in s števili) smo dobili naslednje meritve.

Format	Velikost podatkov s tabelo nizov [B]	Velikost podatkov s tabelo števil [B]
XML	3080	3640
JSON	712	1030
YAML	600	1120
MessagePack	503	503
Bencode	606	1130
Podpora v Javi	560	1090

Tabela 5.3: Meritve velikost podatkov.

Zgornje podatke predstavimo v obliki grafikonu 5.3.Prikazuje velikost datotek, ki vsebujejo serializirane podatke za posamezne formate. Opazimo, da le format XML porabi relativno občutno več prostora. Razlog za to je, ker ne podpira strukturiranih podatkov, zaradi česar serializacija tabele ni optimalna. Zanimiva ugotovitev je format MessagePack potreboval enako prostora za oba testa. Opazimo tudi, da je serializirana tabela števil večja. Razlog je v velikosti elementov tabele - velika števila potrebujejo več prostora kot nizi dolžine štirih črk.



Grafikon 5.3: Velikost podatkov serializirane tabele nizov ter tabele števil.

Poglavje 6

Sklepne ugotovitve

V diplomskem delu je bil narejen pregled tehnologij za serializacijo objektov. Predstavljen je bil proces serializacije in deserializacije ter njun splošen opis. Sledil je opis tehnologij za serializacijo objektov. Opisani so bili formati ASN.1. BSON, MessagePack, Protocol Buffers, Smile, Bencode, JSON, XML, YAML ter podpora serializaciji v Javi kot predstavnik programskih jezikov za interno podporo serializacije. Formati si bili izbrani glede na popularnost in dejstvo, da skupaj pokrivajo čim širše računalniško področje. V drugem delu diplomske naloge je sledilo testiranje formatov, ki je bilo razdeljeno na tri dele – testiranje hitrosti serializacije tabele nizov in tabele števil, testiranje hitrosti deserializacije enake tabele nizov in tabele števil ter testiranje velikosti serializiranih podatkov iz prvih dveh testiranj. Med samim testiranjem ni prišlo do večjih težav. Največ časa sem potreboval za programiranje čim bolj kvalitetne kode, saj je pri testiranju pomembno, da so končni podatki čim bolj standardizirani.

Izkazalo se je, da tekstovni formati potrebujejo več časa za serializacijo in deserializacijo objektov, kot tudi dejstvo, da porabijo serializirani podatki tekstovnih formatov več prostora. Zanimivi so rezultati testiranj za interno podporo v Javi. Pričakoval sem, da bo slednja rešitev hitrejša v hitrosti serializacije in deserializacije od ostalih binarnih formatov, saj je uporabna specifično za objekte, ustvarjene v jeziku Java. Izkazalo se je, da se po rezultatih primerja s formatom Bencode.

Sklepna ugotovitev testov je za posameznika pomembna, ko se odloča, kateri format uporabiti. Če so hitrost serializacije in kompaktnost podatkov pomembni, je ustrezen binarni format. Če je pomembna razumljivost serializiranih podatkov, je ustrezen tekstovni format.

Diplomsko nalogo bi se lahko razširilo oziroma nadaljevalo v veliko smereh. Lahko bi opisal še več formatov, pri čemer bi dobil ustreznejše formate za specifične situacije. Poleg tega bi lahko opravil pri testiranju več testov z več različnimi podatkovnimi strukturami ter večjo količino podatkov, pri čemer bi dobil jasnejše razlike v hitrosti in kompaktnosti formatov.

6.1 Priporočila pri uporabi tehnologij za serializacijo

V tem delu bom predstavil svoja priporočila, kako se odločiti za ustrezen foramt.

Za začetek je pomembna odločitev, ali je bolj primeren binarni format (hitrejša serializacija in kompaktnost podatkov) ali tekstoven format (razumljivost serializiranih podatkov človeku). Med binarnimi formati se odločamo med Bencode, ASN.1, Protocol Buffers, Smile, BSON in MessagePack. Zadnji trije so relevantni le, če imamo opravka s podatki v JSON formatu. V tem primeru priporočam format MessagePack zaradi hitrosti in popolne kompatibilnosti s formatom JSON, sledi BSON in nato Smile, ker uživa najmanj podpore od vseh treh. Če JSON ni del projekta in ni zaželjena uporaba sheme podatkov, priporočam format Bencode.

Če poznamo shemo podatkov, s katerimi bomo imeli opravka in želimo od formata hitrost in kompaktnost, se obrnemo po ASN.1 in Protocol Buffers, pri čemer priporočam slednjega, ker je v večini primerov boljši zaradi lažje uporabe (tudi v programskih jezikih).

Med tekstovnimi formati se odločamo med formati XML, JSON in YAML. Zadnja dva priporočam, ko imamo opravka s strukturiranimi podatki kot so seznam in slovar, pri čemer je JSON hitrejši (posebno, če se uporablja JavaScript), YAML pa razumljivejši za branje, saj ne vsebuje narekovajev, oglatih oklepajev ter ostalih znakov, ki otežujejo branje. XML zaradi generičnosti formata priporočam le pri markiranju nestrukturiranih podatkov (recimo pri spletnih straneh).

Literatura

[1] Chawla Surbhi in Chawla Rama, »Object serialization formats and techniques - a review«, Global journal of computer science and technology, Software and data engineering – ISSN 0975-4172. – Volume 13, Issue 6 version 1.0, 2013

[2] Data serialization. Dostopno na: https://en.wikipedia.org/wiki/Data_serialization

[3] Comparison of data serialization formats. Dostopno na: https://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats

[4] BSON. Dostopno na: http://bsonspec.org/

[5] Unicode. Dostopno na: https://en.wikipedia.org/wiki/Unicode

[6] XML. Dostopno na: http://www.w3.org/XML/

[7] XML. Dostopno na: https://ucilnica.fri.uni-lj.si/pluginfile.php/43109/mod_label/intro/02-XML.pdf

[8] An Introduction to JSON in JavaScript and .NET. Dostopno na: http://msdn.microsoft.com/en-us/library/bb299886.aspx

[9] JSON. Dostopno na: http://json.org/

[10] BSON. Dostopno na: https://en.wikipedia.org/wiki/BSON

[11] An overview of serialization formats.. Dostopno na: http://blog.daniellobato.me/2013/03/an-overview-of-serialization-formats-with-numbers-and-anecdotes/

[12] YAML. Dostopno na: http://www.yaml.org/

[13] MessagePack specification. Dostopno na: https://github.com/msgpack/msgpack/blob/master/spec.md

[14] MessagePack. Dostopno na: http://msgpack.org/

[15] Introduction to ASN.1. Dostopno na: http://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx

[16] Bencode. Dostopno na: https://en.wikipedia.org/wiki/Bencode

[17] Protocol Buffers. Dostopno na: https://developers.google.com/protocol-buffers/

[18] Smile. Dostopno na: https://en.wikipedia.org/wiki/Smile_(Data_Interchange_Format)

Priloga

Koda 18: Testiranje formata XML

```
package test2;

import java.io.File;
import java.io.IOException;
import java.io.StringWriter;
import java.util.Random;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class XMLtest2 {

     public static void main(String[] args) throws
ParserConfigurationException, TransformerException,
SAXException, IOException{
          long zacetniCas, koncniCas;

          zacetniCas= System.nanoTime();
          //testMakeString();
          //testMakeNumbers();
          //testReadString();
          //testReadNumbers();
          koncniCas= System.nanoTime();

          System.out.println("Porabljen čas: " + (koncniCas -
zacetniCas)/1000000 + " ms.");
     }

     static void testMakeString() throws
ParserConfigurationException, TransformerException{
```

```
          DocumentBuilderFactory docFactory =
DocumentBuilderFactory.newInstance();
          DocumentBuilder docBuilder =
docFactory.newDocumentBuilder();

          Document doc = docBuilder.newDocument();
          Element prvo = doc.createElement("seznam");
          doc.appendChild(prvo);

          for (int i = 0; i < 100; i++) {
               Element element = doc.createElement("element");
               element.setAttribute("id",
Integer.toString(i+1));
               element.appendChild(doc.createTextNode("test"));
               prvo.appendChild(element);
          }
          /*
          javax.xml.transform.TransformerFactory tfactory =
TransformerFactory.newInstance();
          javax.xml.transform.Transformer xform =
tfactory.newTransformer();
          javax.xml.transform.Source src= new DOMSource(doc);
          java.io.StringWriter writer = new StringWriter();
          javax.xml.transform.Result result = new
javax.xml.transform.stream.StreamResult(writer);
          xform.transform(src, result);
          System.out.println(writer.toString());*/

          TransformerFactory transformerFactory =
TransformerFactory.newInstance();
          Transformer transformer =
transformerFactory.newTransformer();
          DOMSource source = new DOMSource(doc);
          StreamResult result2 = new StreamResult(new
File("testni.xml"));
          transformer.transform(source, result2);

          /*
          Document doc = docBuilder.newDocument();
          Element prvo = doc.createElement("sola");
          doc.appendChild(prvo);

          Element ucenec = doc.createElement("ucenec");
          ucenec.setAttribute("id", "1");
          prvo.appendChild(ucenec);

          Element ime = doc.createElement("ime");
          ime.appendChild(doc.createTextNode("Valentin"));
          ucenec.appendChild(ime);
```

```
          Element priimek = doc.createElement("priimek");
          priimek.appendChild(doc.createTextNode("Kragelj"));
          ucenec.appendChild(priimek);

          Element diplomska = doc.createElement("diplomska");
          diplomska.appendChild(doc.createTextNode("naslov"));
          ucenec.appendChild(diplomska);
          */
     }

     static void testReadString() throws
ParserConfigurationException, SAXException, IOException{
          File fXmlFile = new File("testni.xml");
          DocumentBuilderFactory dbFactory =
DocumentBuilderFactory.newInstance();
          DocumentBuilder dBuilder =
dbFactory.newDocumentBuilder();
          Document doc = dBuilder.parse(fXmlFile);
          doc.getDocumentElement().normalize();
          //System.out.println("Root element :" +
doc.getDocumentElement().getNodeName());

          NodeList nList = doc.getElementsByTagName("element");
          for (int temp = 0; temp < nList.getLength(); temp++)
{
               Node nNode = nList.item(temp);
               //System.out.println("\nCurrent Element :" +
nNode.getTextContent());
          }
     }

     static void testMakeNumbers() throws IOException,
TransformerException, ParserConfigurationException{
          DocumentBuilderFactory docFactory =
DocumentBuilderFactory.newInstance();
          DocumentBuilder docBuilder =
docFactory.newDocumentBuilder();

          Document doc = docBuilder.newDocument();
          Element prvo = doc.createElement("seznam");
          doc.appendChild(prvo);

          Random rand= new Random();
          for (int i = 0; i < 100; i++) {
               Element element = doc.createElement("element");
               element.setAttribute("id",
Integer.toString(i+1));

     element.appendChild(doc.createTextNode(Integer.toString(ra
nd.nextInt((int) Math.pow(2, 31)))));
```

```
                prvo.appendChild(element);
            }
            /*
            javax.xml.transform.TransformerFactory tfactory =
TransformerFactory.newInstance();
            javax.xml.transform.Transformer xform =
tfactory.newTransformer();
            javax.xml.transform.Source src= new DOMSource(doc);
            java.io.StringWriter writer = new StringWriter();
            javax.xml.transform.Result result = new
javax.xml.transform.stream.StreamResult(writer);
            xform.transform(src, result);
            System.out.println(writer.toString());*/

            TransformerFactory transformerFactory =
TransformerFactory.newInstance();
            Transformer transformer =
transformerFactory.newTransformer();
            DOMSource source = new DOMSource(doc);
            StreamResult result2 = new StreamResult(new
File("testniNumbers.xml"));
            transformer.transform(source, result2);
        }

        static void testReadNumbers() throws
ParserConfigurationException, SAXException, IOException{
            File fXmlFile = new File("testniNumbers.xml");
            DocumentBuilderFactory dbFactory =
DocumentBuilderFactory.newInstance();
            DocumentBuilder dBuilder =
dbFactory.newDocumentBuilder();
            Document doc = dBuilder.parse(fXmlFile);
            doc.getDocumentElement().normalize();
            //System.out.println("Root element :" +
doc.getDocumentElement().getNodeName());

            NodeList nList = doc.getElementsByTagName("element");
            for (int temp = 0; temp < nList.getLength(); temp++)
{
                Node nNode = nList.item(temp);
                //System.out.println("\nCurrent Element :" +
nNode.getTextContent());
            }
        }
}
```

Koda 19: Testiranje formata JSON.

```
package test2;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;
import java.util.Random;

import org.json.simple.*;
import org.json.simple.parser.JSONParser;
import org.json.simple.parser.ParseException;

public class JSONtest2 {

     public static void main(String[] args) throws IOException,
ParseException {
          long zacetniCas, koncniCas;
          zacetniCas= System.nanoTime();
          //testMakeStrings();
          //testMakeNumbers();
          //testRead();
          testReadNumbers();
          koncniCas= System.nanoTime();

          System.out.println("Porabljen čas: " + (koncniCas -
zacetniCas)/1000000 + " ms.");
     }

     static void testMakeStrings() throws IOException{
          JSONObject obj = new JSONObject();
          JSONArray array= new JSONArray();
          for (int i = 0; i < 100; i++) {
               array.add("test");
          }
          obj.put("seznam", array);
          //System.out.println(array.toJSONString());

          FileWriter file = new FileWriter("testni.json");
          file.write(obj.toJSONString());
          file.flush();
          file.close();

          /*
          JSONObject a= new JSONObject();
          JSONObject b= new JSONObject();
          JSONObject c= new JSONObject();
```

```
          c.put("id", "1");
          c.put("ime", "Valentin");
          c.put("priimek", "Kragelj");
          c.put("diplomska", "naslov");
          b.put("ucenec", c);
          a.put("sola", b);
          a.toJSONString();*/
     }

     static void testMakeNumbers() throws IOException{
          JSONObject obj = new JSONObject();
          JSONArray array= new JSONArray();

          Random rand= new Random();
          for (int i = 0; i < 100; i++) {
               array.add(rand.nextInt((int) Math.pow(2, 31)));
          }
          obj.put("seznam", array);
          //System.out.println(array.toJSONString());

          FileWriter file = new
FileWriter("testniNumbers.json");
          file.write(obj.toJSONString());
          file.flush();
          file.close();
     }

     static void testRead() throws FileNotFoundException,
IOException, ParseException{
          JSONParser parser = new JSONParser();
          Object obj = parser.parse(new
FileReader("testni.json"));
          JSONObject jsonObject = (JSONObject) obj;

          JSONArray msg = (JSONArray) jsonObject.get("seznam");
          /*Iterator<String> iterator = msg.iterator();
          while (iterator.hasNext()) {
               System.out.println(iterator.next());
          }*/
     }

     static void testReadNumbers() throws
FileNotFoundException, IOException, ParseException{
          JSONParser parser = new JSONParser();
          Object obj = parser.parse(new
FileReader("testniNumbers.json"));
          JSONObject jsonObject = (JSONObject) obj;

          JSONArray msg = (JSONArray) jsonObject.get("seznam");
          /*Iterator<String> iterator = msg.iterator();
```

```
          while (iterator.hasNext()) {
               System.out.println(iterator.next());
          }*/
     }
}
```

Koda 20: Testiranje YAML formata.

```
package test2;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Random;

import org.yaml.snakeyaml.Yaml;

public class YAMLtest2 {

     public static void main(String[] args) throws IOException
{
          // TODO Auto-generated method stub
          long zacetniCas, koncniCas;
          zacetniCas= System.nanoTime();
          //testMake();
          //testMakeNumbers();
          testRead();
          //testReadNumbers();
          koncniCas= System.nanoTime();

          System.out.println("Porabljen čas: " + (koncniCas -
zacetniCas)/1000000 + " ms.");

     }

     static void testMake() throws IOException{
          String s= "";
          for (int i = 0; i < 100; i++) {
               s += "\n- test";
          }

          Yaml yaml = new Yaml();
          List<String> list = (List<String>) yaml.load(s);
          System.out.println(list.toString());
```

```
          FileWriter file = new FileWriter("testni.yaml");
          file.write(list.toString());
          file.flush();
          file.close();
     }

     static void testMakeNumbers() throws IOException{ //YAML
autodetects the datatype of the entity
          String s= "";
          Random rand = new Random();
          for (int i = 0; i < 100; i++) {
               s += "\n- " + rand.nextInt((int) Math.pow(2,
31));
          }

          Yaml yaml = new Yaml();
          List<String> list = (List<String>) yaml.load(s);
          //System.out.println(list.toString());

          FileWriter file = new
FileWriter("testniNumbers.yaml");
          file.write(list.toString());
          file.flush();
          file.close();
     }

static void testRead(){
     final Yaml yaml = new Yaml();
      Reader reader = null;
      LinkedHashMap lhm = new LinkedHashMap();

      try {
          reader = new FileReader("testni.yaml");
          ArrayList<String> deser= (ArrayList<String>)
yaml.load(reader);
          //System.out.println(deser);
      } catch (final FileNotFoundException e) {
          System.err.println("Datoteka ne obstaja: " + e);
      } finally {
          if (null != reader) {
              try {
                  reader.close();
              } catch (final IOException e) {
                  System.err.println("Napaka na koncu: " + e);
              }
          }
      }
}
```

```
     static void testReadNumbers(){
          final Yaml yaml = new Yaml();
           Reader reader = null;
           LinkedHashMap lhm = new LinkedHashMap();

           try {
               reader = new FileReader("testniNumbers.yaml");

               //System.out.println("yaml: "+
yaml.load(reader));
           } catch (final FileNotFoundException fnfe) {
               System.err.println("We had a problem reading the
YAML from the file because we couldn't find the file." +
fnfe);
           } finally {
               if (null != reader) {
                   try {
                       reader.close();
                   } catch (final IOException ioe) {
                       System.err.println("We got the following
exception trying to clean up the reader: " + ioe);
                   }
               }
           }
     }
}
```

Koda 21: Testiranje MessagePack formata.

```
package test;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import org.msgpack.MessagePack;
import org.msgpack.template.Templates;

public class MsgPackTest {

     public static void main(String[] args) throws IOException
{
          // TODO Auto-generated method stub
```

```
          long zacetniCas, koncniCas;

          zacetniCas= System.nanoTime();
          //testMake();
          //testMakeNumbers();
          //testRead();
          testReadNumber();
          koncniCas= System.nanoTime();

          System.out.println("Porabljen čas: " + (koncniCas -
zacetniCas)/1000000 + " ms.");
     }

     static void testMake() throws IOException{
          List<String> seznam = new ArrayList<String>();
          for (int i = 0; i < 100; i++) {
               seznam.add("test");
          }
          MessagePack msgpack = new MessagePack();
          /*byte[] ser = msgpack.write(seznam); //serializacija


          List<String> deser = msgpack.read(ser,
Templates.tList(Templates.tString()));
          for (int i = 0; i < deser.size(); i++) {
               System.out.println(deser.get(i));
          }*/

          FileOutputStream output = new FileOutputStream(new
File("testni.msgPack"));
          msgpack.write(output, seznam);
     }

     static void testMakeNumbers() throws IOException{
          List<Integer> seznam = new ArrayList<Integer>();
          Random rand= new Random();
          for (int i = 0; i < 100; i++) {
               seznam.add(rand.nextInt((int) Math.pow(2, 31)));
          }
          MessagePack msgpack = new MessagePack();

          FileOutputStream output = new FileOutputStream(new
File("testniNumbers.msgPack"));
          msgpack.write(output, seznam);
     }

     static void testRead() throws IOException{
          MessagePack msgpack = new MessagePack();
```

```
          FileInputStream input = new FileInputStream(new
File("testni.msgPack"));
          List<String> bla = msgpack.read(input,
Templates.tList(Templates.tString()));
          /*for (int i = 0; i < bla.size(); i++) {
               System.out.println(bla.get(i));
          }*/
     }

     static void testReadNumber() throws IOException{
          MessagePack msgpack = new MessagePack();

          FileInputStream input = new FileInputStream(new
File("testniNumbers.msgPack"));
          List<Integer> bla = msgpack.read(input,
Templates.tList(Templates.tInteger()));
          /*for (int i = 0; i < bla.size(); i++) {
               System.out.println(bla.get(i));
          }*/
     }
}
```

Koda 22: Testiranje serializacije Bencode formata.

```
package test;

import java.io.ByteArrayOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.SortedSet;

import com.googlecode.jbencode.composite.DictionaryType;
import com.googlecode.jbencode.composite.EntryType;
import com.googlecode.jbencode.composite.ListType;
import com.googlecode.jbencode.composite.ListTypeStream;
import com.googlecode.jbencode.primitive.*;

public class BencodeTest {

     public static void main(String[] args) throws IOException
{
          // TODO Auto-generated method stub
          long zacetniCas, koncniCas;

          zacetniCas= System.nanoTime();
          //testMake();
```

```
          testMakeNumbers();
          koncniCas= System.nanoTime();

          System.out.println("Porabljen čas: " + (koncniCas -
zacetniCas)/1000000 + " ms.");
     }

     static void testMake() throws IOException{
          ByteArrayOutputStream os = new
ByteArrayOutputStream();

          DictionaryType root = new DictionaryType() {
               @Override
               protected void populate(SortedSet<EntryType<?>>
entries) {
                    // TODO Auto-generated method stub
                    /*for (int i = 0; i < 50; i++) {
                         entries.add(new
EntryType<LiteralStringType>(new LiteralStringType("test"),
new LiteralStringType("test")));
                    }*/

                    entries.add(new
EntryType<LiteralStringType>(
                         new LiteralStringType(""),
                         new ListType() {

                     @Override
                     protected void populate(ListTypeStream
list) throws IOException {
                         for (int i = 0; i < 100; i++) {
                              list.add(new
LiteralStringType("test"));
                         }
                     }
                  }));
               }};

          root.write(os);

           System.out.println(new String(os.toByteArray()));

           FileWriter file = new FileWriter("testni.bencode");
           file.write(new String(os.toByteArray()));
           file.flush();
           file.close();
     }

     static void testMakeNumbers() throws IOException{
```

```
          ByteArrayOutputStream os = new
ByteArrayOutputStream();

          DictionaryType root = new DictionaryType() {
               @Override
               protected void populate(SortedSet<EntryType<?>>
entries) {
                    // TODO Auto-generated method stub
                    for (int i = 0; i < 50; i++) {
                         //entries.add(new
EntryType<LiteralStringType>(new LiteralStringType("test"),
new LiteralStringType("test")));
                    }

                    entries.add(new
EntryType<LiteralStringType>(
                         new LiteralStringType(""),
                         new ListType() {

                     @Override
                     protected void populate(ListTypeStream
list) throws IOException {
                         Random rand= new Random();
                         for (int i = 0; i < 100; i++) {
                              list.add(new
IntegerType(rand.nextInt((int) Math.pow(2, 31))));
                         }
                     }
                  }));
               }};

          root.write(os);

           //System.out.println(new String(os.toByteArray()));

           FileWriter file = new
FileWriter("testniNumbers.bencode");
           file.write(new String(os.toByteArray()));
           file.flush();
           file.close();
     }

     private static class LiteralStringType extends StringType
implements Comparable<LiteralStringType> {
          private final String value;

          public LiteralStringType(String value) {
              this.value = value;
          }
```

```
        @Override
        protected long getLength() {
            return value.length();
        }

        @Override
        protected void writeValue(OutputStream os) throws
IOException {
            os.write(value.getBytes("US-ASCII"));
        }

        public int compareTo(LiteralStringType o) {
            return o.value.compareTo(value);
        }
    }
}
```

Koda 23: Testiranje deserializacije Bencode formata.

```
package org.pixie.bencoding;

import org.pixie.bencoding.bencode.BElement;
import org.pixie.bencoding.bencode.BReader;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;

/**
 * Project Bencoding
 * Created by Francis on 26/03/14.
 */
public class Application {

    public static void main(final String[] args) {
     long zacetniCas, koncniCas;

          zacetniCas= System.nanoTime();
          //testRead();
          testReadNumbers();
          koncniCas= System.nanoTime();

          System.out.println("Porabljen čas: " + (koncniCas -
zacetniCas)/1000000 + " ms.");
    }

    static void testRead(){
     String s= "";
```

```
     BufferedReader br = null;
          try {
               String sCurrentLine;
               br = new BufferedReader(new
FileReader("testni.bencode"));
               while ((sCurrentLine = br.readLine()) != null) {
                    //System.out.println(sCurrentLine);
                    s= sCurrentLine;
               }
          } catch (IOException e) {
               e.printStackTrace();
          } finally {
               try {
                    if (br != null)
                         br.close();
               } catch (IOException ex) {
                    ex.printStackTrace();
               }
          }

        final BElement[] elements3 = BReader.read(s);
        //System.out.println(Arrays.toString(elements3));
    }

    static void testReadNumbers(){
     String s= "";
     BufferedReader br = null;
          try {
               String sCurrentLine;
               br = new BufferedReader(new
FileReader("testniNumbers.bencode"));
               while ((sCurrentLine = br.readLine()) != null) {
                    //System.out.println(sCurrentLine);
                    s= sCurrentLine;
               }
          } catch (IOException e) {
               e.printStackTrace();
          } finally {
               try {
                    if (br != null)
                         br.close();
               } catch (IOException ex) {
                    ex.printStackTrace();
               }
          }

        final BElement[] elements3 = BReader.read(s);
        //System.out.println(Arrays.toString(elements3));
    }
}
```

Koda 24: Testiranje interne podpora serializacije jezika Java.

```
package serializacijatest;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class SerializacijaTest {

    public static void main(String[] args) throws IOException,
ClassNotFoundException {

        long zacetniCas, koncniCas;

        zacetniCas= System.nanoTime();
        //testMake();
        //testMakeNumbers();
        //testRead();
        testReadNumbers();
        koncniCas= System.nanoTime();

        System.out.println("Porabljen čas: " + (koncniCas -
zacetniCas)/1000000 + " ms.");
    }

    static void testMake() throws IOException,
ClassNotFoundException{
        List<String> seznam = new ArrayList<String>();
        for (int i = 0; i < 100; i++) {
            seznam.add("test");
        }
        ObjectOutputStream ser = new ObjectOutputStream(new
FileOutputStream("serializacijaJava.ser"));
        ser.writeObject(seznam);
        //System.out.println("Serializacija koncana.");
    }

    static void testMakeNumbers() throws IOException,
ClassNotFoundException{
        List<Integer> seznam = new ArrayList<Integer>();
        Random rand= new Random();
```

```
        for (int i = 0; i < 100; i++) {
            seznam.add(rand.nextInt((int) Math.pow(2, 31)));
        }
        ObjectOutputStream ser = new ObjectOutputStream(new
FileOutputStream("serializacijaNumbersJava.ser"));
        ser.writeObject(seznam);
        //System.out.println("Serializacija koncana.");
    }

    static void testRead() throws FileNotFoundException,
IOException, ClassNotFoundException{
        ObjectInputStream deser = new ObjectInputStream( new
FileInputStream("serializacijaJava.ser") );
        List<String> seznam = new ArrayList<String>();
        seznam = (List<String>) deser.readObject();
        //System.out.println(seznam);
    }

    static void testReadNumbers() throws
FileNotFoundException, IOException, ClassNotFoundException{
        ObjectInputStream deser = new ObjectInputStream( new
FileInputStream("serializacijaNumbersJava.ser") );
        List<Integer> seznam = new ArrayList<Integer>();
        seznam = (List<Integer>) deser.readObject();
        //System.out.println(seznam);
    }
}
```